The Python Library Reference

Release 2.7.16

Guido van Rossum
and the Python development team

March 02, 2019

Python Software Foundation
Email: docs@python.org

CONTENTS

1 Introduction 3

2 Built-in Functions 5

3 Non-essential Built-in Functions 27

4 Built-in Constants 29
4.1 Constants added by the site module . . . 29

5 Built-in Types 31
5.1 Truth Value Testing . . . 31
5.2 Boolean Operations — and, or, not . . . 32
5.3 Comparisons . . . 32
5.4 Numeric Types — int, float, long, complex . . . 33
5.5 Iterator Types . . . 37
5.6 Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange . . . 37
5.7 Set Types — set, frozenset . . . 50
5.8 Mapping Types — dict . . . 53
5.9 File Objects . . . 57
5.10 memoryview type . . . 61
5.11 Context Manager Types . . . 62
5.12 Other Built-in Types . . . 63
5.13 Special Attributes . . . 66

6 Built-in Exceptions 67
6.1 Exception hierarchy . . . 72

7 String Services 75
7.1 string — Common string operations . . . 75
7.2 re — Regular expression operations . . . 88
7.3 struct — Interpret strings as packed binary data . . . 104
7.4 difflib — Helpers for computing deltas . . . 109
7.5 StringIO — Read and write strings as files . . . 120
7.6 cStringIO — Faster version of StringIO . . . 121
7.7 textwrap — Text wrapping and filling . . . 122
7.8 codecs — Codec registry and base classes . . . 124
7.9 unicodedata — Unicode Database . . . 139
7.10 stringprep — Internet String Preparation . . . 141
7.11 fpformat — Floating point conversions . . . 142

8 Data Types 145

8.1 datetime — Basic date and time types . . . 145
8.2 calendar — General calendar-related functions . . . 170
8.3 collections — High-performance container datatypes . . . 174
8.4 heapq — Heap queue algorithm . . . 189
8.5 bisect — Array bisection algorithm . . . 193
8.6 array — Efficient arrays of numeric values . . . 195
8.7 sets — Unordered collections of unique elements . . . 198
8.8 sched — Event scheduler . . . 201
8.9 mutex — Mutual exclusion support . . . 203
8.10 Queue — A synchronized queue class . . . 204
8.11 weakref — Weak references . . . 206
8.12 UserDict — Class wrapper for dictionary objects . . . 210
8.13 UserList — Class wrapper for list objects . . . 211
8.14 UserString — Class wrapper for string objects . . . 212
8.15 types — Names for built-in types . . . 213
8.16 new — Creation of runtime internal objects . . . 216
8.17 copy — Shallow and deep copy operations . . . 216
8.18 pprint — Data pretty printer . . . 217
8.19 repr — Alternate repr() implementation . . . 220

9 Numeric and Mathematical Modules 223
9.1 numbers — Numeric abstract base classes . . . 223
9.2 math — Mathematical functions . . . 226
9.3 cmath — Mathematical functions for complex numbers . . . 230
9.4 decimal — Decimal fixed point and floating point arithmetic . . . 233
9.5 fractions — Rational numbers . . . 260
9.6 random — Generate pseudo-random numbers . . . 262
9.7 itertools — Functions creating iterators for efficient looping . . . 266
9.8 functools — Higher-order functions and operations on callable objects . . . 279
9.9 operator — Standard operators as functions . . . 282

10 File and Directory Access 291
10.1 os.path — Common pathname manipulations . . . 291
10.2 fileinput — Iterate over lines from multiple input streams . . . 295
10.3 stat — Interpreting stat() results . . . 297
10.4 statvfs — Constants used with os.statvfs() . . . 301
10.5 filecmp — File and Directory Comparisons . . . 302
10.6 tempfile — Generate temporary files and directories . . . 304
10.7 glob — Unix style pathname pattern expansion . . . 307
10.8 fnmatch — Unix filename pattern matching . . . 308
10.9 linecache — Random access to text lines . . . 309
10.10 shutil — High-level file operations . . . 309
10.11 dircache — Cached directory listings . . . 314
10.12 macpath — Mac OS 9 path manipulation functions . . . 315

11 Data Persistence 317
11.1 pickle — Python object serialization . . . 317
11.2 cPickle — A faster pickle . . . 328
11.3 copy_reg — Register pickle support functions . . . 328
11.4 shelve — Python object persistence . . . 329
11.5 marshal — Internal Python object serialization . . . 331
11.6 anydbm — Generic access to DBM-style databases . . . 333
11.7 whichdb — Guess which DBM module created a database . . . 334
11.8 dbm — Simple “database” interface . . . 335

11.9 gdbm — GNU's reinterpretation of dbm . . . 335
11.10 dbhash — DBM-style interface to the BSD database library . . . 337
11.11 bsddb — Interface to Berkeley DB library . . . 338
11.12 dumbdbm — Portable DBM implementation . . . 341
11.13 sqlite3 — DB-API 2.0 interface for SQLite databases . . . 342

12 Data Compression and Archiving 363
12.1 zlib — Compression compatible with gzip . . . 363
12.2 gzip — Support for gzip files . . . 366
12.3 bz2 — Compression compatible with bzip2 . . . 368
12.4 zipfile — Work with ZIP archives . . . 370
12.5 tarfile — Read and write tar archive files . . . 376

13 File Formats 387
13.1 csv — CSV File Reading and Writing . . . 387
13.2 ConfigParser — Configuration file parser . . . 395
13.3 robotparser — Parser for robots.txt . . . 402
13.4 netrc — netrc file processing . . . 403
13.5 xdrlib — Encode and decode XDR data . . . 404
13.6 plistlib — Generate and parse Mac OS X .plist files . . . 407

14 Cryptographic Services 409
14.1 hashlib — Secure hashes and message digests . . . 409
14.2 hmac — Keyed-Hashing for Message Authentication . . . 411
14.3 md5 — MD5 message digest algorithm . . . 413
14.4 sha — SHA-1 message digest algorithm . . . 414

15 Generic Operating System Services 415
15.1 os — Miscellaneous operating system interfaces . . . 415
15.2 io — Core tools for working with streams . . . 445
15.3 time — Time access and conversions . . . 456
15.4 argparse — Parser for command-line options, arguments and sub-commands . . . 462
15.5 optparse — Parser for command line options . . . 492
15.6 getopt — C-style parser for command line options . . . 519
15.7 logging — Logging facility for Python . . . 521
15.8 logging.config — Logging configuration . . . 535
15.9 logging.handlers — Logging handlers . . . 545
15.10 getpass — Portable password input . . . 553
15.11 curses — Terminal handling for character-cell displays . . . 554
15.12 curses.textpad — Text input widget for curses programs . . . 571
15.13 curses.ascii — Utilities for ASCII characters . . . 573
15.14 curses.panel — A panel stack extension for curses . . . 575
15.15 platform — Access to underlying platform's identifying data . . . 576
15.16 errno — Standard errno system symbols . . . 580
15.17 ctypes — A foreign function library for Python . . . 586

16 Optional Operating System Services 621
16.1 select — Waiting for I/O completion . . . 621
16.2 threading — Higher-level threading interface . . . 626
16.3 thread — Multiple threads of control . . . 637
16.4 dummy_threading — Drop-in replacement for the threading module . . . 638
16.5 dummy_thread — Drop-in replacement for the thread module . . . 639
16.6 multiprocessing — Process-based "threading" interface . . . 639
16.7 mmap — Memory-mapped file support . . . 692
16.8 readline — GNU readline interface . . . 695

16.9 rlcompleter — Completion function for GNU readline . . . 699

17 Interprocess Communication and Networking 701
17.1 subprocess — Subprocess management . . . 701
17.2 socket — Low-level networking interface . . . 713
17.3 ssl — TLS/SSL wrapper for socket objects . . . 726
17.4 signal — Set handlers for asynchronous events . . . 751
17.5 popen2 — Subprocesses with accessible I/O streams . . . 754
17.6 asyncore — Asynchronous socket handler . . . 756
17.7 asynchat — Asynchronous socket command/response handler . . . 760

18 Internet Data Handling 765
18.1 email — An email and MIME handling package . . . 765
18.2 json — JSON encoder and decoder . . . 797
18.3 mailcap — Mailcap file handling . . . 805
18.4 mailbox — Manipulate mailboxes in various formats . . . 806
18.5 mhlib — Access to MH mailboxes . . . 826
18.6 mimetools — Tools for parsing MIME messages . . . 828
18.7 mimetypes — Map filenames to MIME types . . . 829
18.8 MimeWriter — Generic MIME file writer . . . 832
18.9 mimify — MIME processing of mail messages . . . 833
18.10 multifile — Support for files containing distinct parts . . . 834
18.11 rfc822 — Parse RFC 2822 mail headers . . . 836
18.12 base64 — RFC 3548: Base16, Base32, Base64 Data Encodings . . . 840
18.13 binhex — Encode and decode binhex4 files . . . 842
18.14 binascii — Convert between binary and ASCII . . . 842
18.15 quopri — Encode and decode MIME quoted-printable data . . . 844
18.16 uu — Encode and decode uuencode files . . . 845

19 Structured Markup Processing Tools 847
19.1 HTMLParser — Simple HTML and XHTML parser . . . 847
19.2 sgmllib — Simple SGML parser . . . 852
19.3 htmllib — A parser for HTML documents . . . 855
19.4 htmlentitydefs — Definitions of HTML general entities . . . 856
19.5 XML Processing Modules . . . 857
19.6 XML vulnerabilities . . . 858
19.7 xml.etree.ElementTree — The ElementTree XML API . . . 859
19.8 xml.dom — The Document Object Model API . . . 872
19.9 xml.dom.minidom — Minimal DOM implementation . . . 883
19.10 xml.dom.pulldom — Support for building partial DOM trees . . . 887
19.11 xml.sax — Support for SAX2 parsers . . . 888
19.12 xml.sax.handler — Base classes for SAX handlers . . . 890
19.13 xml.sax.saxutils — SAX Utilities . . . 895
19.14 xml.sax.xmlreader — Interface for XML parsers . . . 896
19.15 xml.parsers.expat — Fast XML parsing using Expat . . . 900

20 Internet Protocols and Support 911
20.1 webbrowser — Convenient Web-browser controller . . . 911
20.2 cgi — Common Gateway Interface support . . . 914
20.3 cgitb — Traceback manager for CGI scripts . . . 921
20.4 wsgiref — WSGI Utilities and Reference Implementation . . . 921
20.5 urllib — Open arbitrary resources by URL . . . 930
20.6 urllib2 — extensible library for opening URLs . . . 937
20.7 httplib — HTTP protocol client . . . 949
20.8 ftplib — FTP protocol client . . . 956

20.9 poplib — POP3 protocol client . . . 960
20.10 imaplib — IMAP4 protocol client . . . 962
20.11 nntplib — NNTP protocol client . . . 968
20.12 smtplib — SMTP protocol client . . . 972
20.13 smtpd — SMTP Server . . . 977
20.14 telnetlib — Telnet client . . . 978
20.15 uuid — UUID objects according to RFC 4122 . . . 981
20.16 urlparse — Parse URLs into components . . . 984
20.17 SocketServer — A framework for network servers . . . 988
20.18 BaseHTTPServer — Basic HTTP server . . . 996
20.19 SimpleHTTPServer — Simple HTTP request handler . . . 1000
20.20 CGIHTTPServer — CGI-capable HTTP request handler . . . 1001
20.21 cookielib — Cookie handling for HTTP clients . . . 1002
20.22 Cookie — HTTP state management . . . 1011
20.23 xmlrpclib — XML-RPC client access . . . 1015
20.24 SimpleXMLRPCServer — Basic XML-RPC server . . . 1023
20.25 DocXMLRPCServer — Self-documenting XML-RPC server . . . 1027

21 Multimedia Services 1029
21.1 audioop — Manipulate raw audio data . . . 1029
21.2 imageop — Manipulate raw image data . . . 1032
21.3 aifc — Read and write AIFF and AIFC files . . . 1033
21.4 sunau — Read and write Sun AU files . . . 1035
21.5 wave — Read and write WAV files . . . 1038
21.6 chunk — Read IFF chunked data . . . 1040
21.7 colorsys — Conversions between color systems . . . 1041
21.8 imghdr — Determine the type of an image . . . 1042
21.9 sndhdr — Determine type of sound file . . . 1043
21.10 ossaudiodev — Access to OSS-compatible audio devices . . . 1044

22 Internationalization 1049
22.1 gettext — Multilingual internationalization services . . . 1049
22.2 locale — Internationalization services . . . 1059

23 Program Frameworks 1067
23.1 cmd — Support for line-oriented command interpreters . . . 1067
23.2 shlex — Simple lexical analysis . . . 1069

24 Graphical User Interfaces with Tk 1075
24.1 Tkinter — Python interface to Tcl/Tk . . . 1075
24.2 ttk — Tk themed widgets . . . 1086
24.3 Tix — Extension widgets for Tk . . . 1104
24.4 ScrolledText — Scrolled Text Widget . . . 1109
24.5 turtle — Turtle graphics for Tk . . . 1110
24.6 IDLE . . . 1140
24.7 Other Graphical User Interface Packages . . . 1148

25 Development Tools 1151
25.1 pydoc — Documentation generator and online help system . . . 1151
25.2 doctest — Test interactive Python examples . . . 1152
25.3 unittest — Unit testing framework . . . 1177
25.4 2to3 - Automated Python 2 to 3 code translation . . . 1202
25.5 test — Regression tests package for Python . . . 1207
25.6 test.support — Utility functions for tests . . . 1210

26 Debugging and Profiling 1215
26.1 bdb — Debugger framework . . . 1215
26.2 pdb — The Python Debugger . . . 1219
26.3 Debugger Commands . . . 1221
26.4 The Python Profilers . . . 1224
26.5 hotshot — High performance logging profiler . . . 1233
26.6 timeit — Measure execution time of small code snippets . . . 1234
26.7 trace — Trace or track Python statement execution . . . 1239

27 Software Packaging and Distribution 1243
27.1 distutils — Building and installing Python modules . . . 1243
27.2 ensurepip — Bootstrapping the pip installer . . . 1243

28 Python Runtime Services 1247
28.1 sys — System-specific parameters and functions . . . 1247
28.2 sysconfig — Provide access to Python's configuration information . . . 1259
28.3 __builtin__ — Built-in objects . . . 1262
28.4 future_builtins — Python 3 builtins . . . 1263
28.5 __main__ — Top-level script environment . . . 1264
28.6 warnings — Warning control . . . 1264
28.7 contextlib — Utilities for with-statement contexts . . . 1269
28.8 abc — Abstract Base Classes . . . 1271
28.9 atexit — Exit handlers . . . 1274
28.10 traceback — Print or retrieve a stack traceback . . . 1276
28.11 __future__ — Future statement definitions . . . 1279
28.12 gc — Garbage Collector interface . . . 1281
28.13 inspect — Inspect live objects . . . 1283
28.14 site — Site-specific configuration hook . . . 1289
28.15 user — User-specific configuration hook . . . 1292
28.16 fpectl — Floating point exception control . . . 1293

29 Custom Python Interpreters 1295
29.1 code — Interpreter base classes . . . 1295
29.2 codeop — Compile Python code . . . 1297

30 Restricted Execution 1299
30.1 rexec — Restricted execution framework . . . 1299
30.2 Bastion — Restricting access to objects . . . 1303

31 Importing Modules 1305
31.1 imp — Access the import internals . . . 1305
31.2 importlib — Convenience wrappers for __import__() . . . 1308
31.3 imputil — Import utilities . . . 1309
31.4 zipimport — Import modules from Zip archives . . . 1312
31.5 pkgutil — Package extension utility . . . 1314
31.6 modulefinder — Find modules used by a script . . . 1316
31.7 runpy — Locating and executing Python modules . . . 1318

32 Python Language Services 1321
32.1 parser — Access Python parse trees . . . 1321
32.2 ast — Abstract Syntax Trees . . . 1325
32.3 symtable — Access to the compiler's symbol tables . . . 1330
32.4 symbol — Constants used with Python parse trees . . . 1332
32.5 token — Constants used with Python parse trees . . . 1333
32.6 keyword — Testing for Python keywords . . . 1334

32.7 tokenize — Tokenizer for Python source . . . 1335
32.8 tabnanny — Detection of ambiguous indentation . . . 1337
32.9 pyclbr — Python class browser support . . . 1337
32.10 py_compile — Compile Python source files . . . 1339
32.11 compileall — Byte-compile Python libraries . . . 1339
32.12 dis — Disassembler for Python bytecode . . . 1341
32.13 pickletools — Tools for pickle developers . . . 1350

33 Python compiler package 1351
33.1 The basic interface . . . 1351
33.2 Limitations . . . 1352
33.3 Python Abstract Syntax . . . 1352
33.4 Using Visitors to Walk ASTs . . . 1357
33.5 Bytecode Generation . . . 1358

34 Miscellaneous Services 1359
34.1 formatter — Generic output formatting . . . 1359

35 MS Windows Specific Services 1365
35.1 msilib — Read and write Microsoft Installer files . . . 1365
35.2 msvcrt — Useful routines from the MS VC++ runtime . . . 1370
35.3 _winreg — Windows registry access . . . 1372
35.4 winsound — Sound-playing interface for Windows . . . 1380

36 Unix Specific Services 1383
36.1 posix — The most common POSIX system calls . . . 1383
36.2 pwd — The password database . . . 1384
36.3 spwd — The shadow password database . . . 1385
36.4 grp — The group database . . . 1385
36.5 crypt — Function to check Unix passwords . . . 1386
36.6 dl — Call C functions in shared objects . . . 1387
36.7 termios — POSIX style tty control . . . 1388
36.8 tty — Terminal control functions . . . 1389
36.9 pty — Pseudo-terminal utilities . . . 1390
36.10 fcntl — The fcntl and ioctl system calls . . . 1390
36.11 pipes — Interface to shell pipelines . . . 1392
36.12 posixfile — File-like objects with locking support . . . 1394
36.13 resource — Resource usage information . . . 1396
36.14 nis — Interface to Sun's NIS (Yellow Pages) . . . 1399
36.15 syslog — Unix syslog library routines . . . 1399
36.16 commands — Utilities for running commands . . . 1401

37 Mac OS X specific services 1403
37.1 ic — Access to the Mac OS X Internet Config . . . 1403
37.2 MacOS — Access to Mac OS interpreter features . . . 1404
37.3 macostools — Convenience routines for file manipulation . . . 1406
37.4 findertools — The finder's Apple Events interface . . . 1407
37.5 EasyDialogs — Basic Macintosh dialogs . . . 1407
37.6 FrameWork — Interactive application framework . . . 1410
37.7 autoGIL — Global Interpreter Lock handling in event loops . . . 1413
37.8 Mac OS Toolbox Modules . . . 1414
37.9 ColorPicker — Color selection dialog . . . 1417

38 MacPython OSA Modules 1419
38.1 gensuitemodule — Generate OSA stub packages . . . 1420

38.2 aetools — OSA client support . . . 1421
38.3 aepack — Conversion between Python variables and AppleEvent data containers . . . 1422
38.4 aetypes — AppleEvent objects . . . 1423
38.5 MiniAEFrame — Open Scripting Architecture server support . . . 1424

39 SGI IRIX Specific Services 1427
39.1 al — Audio functions on the SGI . . . 1427
39.2 AL — Constants used with the al module . . . 1429
39.3 cd — CD-ROM access on SGI systems . . . 1429
39.4 fl — FORMS library for graphical user interfaces . . . 1433
39.5 FL — Constants used with the fl module . . . 1437
39.6 flp — Functions for loading stored FORMS designs . . . 1438
39.7 fm — Font Manager interface . . . 1438
39.8 gl — Graphics Library interface . . . 1439
39.9 DEVICE — Constants used with the gl module . . . 1441
39.10 GL — Constants used with the gl module . . . 1441
39.11 imgfile — Support for SGI imglib files . . . 1441
39.12 jpeg — Read and write JPEG files . . . 1442

40 SunOS Specific Services 1445
40.1 sunaudiodev — Access to Sun audio hardware . . . 1445
40.2 SUNAUDIODEV — Constants used with sunaudiodev . . . 1446

41 Undocumented Modules 1447
41.1 Miscellaneous useful utilities . . . 1447
41.2 Platform specific modules . . . 1447
41.3 Multimedia . . . 1447
41.4 Undocumented Mac OS modules . . . 1448
41.5 Obsolete . . . 1449
41.6 SGI-specific Extension modules . . . 1449

A Glossary 1451

B About these documents 1461
B.1 Contributors to the Python Documentation . . . 1461

C History and License 1463
C.1 History of the software . . . 1463
C.2 Terms and conditions for accessing or otherwise using Python . . . 1464
C.3 Licenses and Acknowledgements for Incorporated Software . . . 1467

D Copyright 1479

Bibliography 1481

Python Module Index 1483

Index 1489

While reference-index describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

Python's standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

In addition to the standard library, there is a growing collection of several thousand components (from individual programs and modules to packages and entire application development frameworks), available from the Python Package Index.

CHAPTER

ONE

INTRODUCTION

The "Python library" contains several different kinds of components.

It contains data types that would normally be considered part of the "core" of a language, such as numbers and lists. For these types, the Python language core defines the form of literals and places some constraints on their semantics, but does not fully define the semantics. (On the other hand, the language core does define syntactic properties like the spelling and priorities of operators.)

The library also contains built-in functions and exceptions — objects that can be used by all Python code without the need of an import statement. Some of these are defined by the core language, but many are not essential for the core semantics and are only described here.

The bulk of the library, however, consists of a collection of modules. There are many ways to dissect this collection. Some modules are written in C and built in to the Python interpreter; others are written in Python and imported in source form. Some modules provide interfaces that are highly specific to Python, like printing a stack trace; some provide interfaces that are specific to particular operating systems, such as access to specific hardware; others provide interfaces that are specific to a particular application domain, like the World Wide Web. Some modules are available in all versions and ports of Python; others are only available when the underlying system supports or requires them; yet others are available only when a particular configuration option was chosen at the time when Python was compiled and installed.

This manual is organized "from the inside out:" it first describes the built-in data types, then the built-in functions and exceptions, and finally the modules, grouped in chapters of related modules. The ordering of the chapters as well as the ordering of the modules within each chapter is roughly from most relevant to least important.

This means that if you start reading this manual from the start, and skip to the next chapter when you get bored, you will get a reasonable overview of the available modules and application areas that are supported by the Python library. Of course, you don't have to read it like a novel — you can also browse the table of contents (in front of the manual), or look for a specific function, module or term in the index (in the back). And finally, if you enjoy learning about random subjects, you choose a random page number (see module random) and read a section or two. Regardless of the order in which you read the sections of this manual, it helps to start with chapter Built-in Functions, as the remainder of the manual assumes familiarity with this material.

Let the show begin!

CHAPTER

TWO

BUILT-IN FUNCTIONS

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	
delattr()	help()	next()	setattr()	
dict()	hex()	object()	slice()	
dir()	id()	oct()	sorted()	

In addition, there are other four built-in functions that are no longer considered essential: apply(), buffer(), coerce(), and intern(). They are documented in the Non-essential Built-in Functions section.

abs(x)
: Return the absolute value of a number. The argument may be a plain or long integer or a floating point number. If the argument is a complex number, its magnitude is returned.

all(iterable)
: Return True if all elements of the iterable are true (or if the iterable is empty). Equivalent to:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

New in version 2.5.

any(iterable)
: Return True if any element of the iterable is true. If the iterable is empty, return False. Equivalent to:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

New in version 2.5.

basestring()
: This abstract type is the superclass for str and unicode. It cannot be called or instantiated, but it can be used to test whether an object is an instance of str or unicode. isinstance(obj, basestring) is equivalent to isinstance(obj, (str, unicode)).

  New in version 2.3.

bin(x)
: Convert an integer number to a binary string. The result is a valid Python expression. If x is not a Python int object, it has to define an __index__() method that returns an integer.

  New in version 2.6.

class bool([x])
: Return a Boolean value, i.e. one of True or False. x is converted using the standard truth testing procedure. If x is false or omitted, this returns False; otherwise it returns True. bool is also a class, which is a subclass of int. Class bool cannot be subclassed further. Its only instances are False and True.

  New in version 2.2.1.

  Changed in version 2.3: If no argument is given, this function returns False.

class bytearray([source[, encoding[, errors]]])
: Return a new array of bytes. The bytearray class is a mutable sequence of integers in the range 0 <= x < 256. It has most of the usual methods of mutable sequences, described in Mutable Sequence Types, as well as most methods that the str type has, see String Methods.

  The optional source parameter can be used to initialize the array in a few different ways:

  - If it is unicode, you must also give the encoding (and optionally, errors) parameters; bytearray() then converts the unicode to bytes using unicode.encode().
  - If it is an integer, the array will have that size and will be initialized with null bytes.
  - If it is an object conforming to the buffer interface, a read-only buffer of the object will be used to initialize the bytes array.
  - If it is an iterable, it must be an iterable of integers in the range 0 <= x < 256, which are used as the initial contents of the array.

  Without an argument, an array of size 0 is created.

  New in version 2.6.

callable(object)
: Return True if the object argument appears callable, False if not. If this returns true, it is still possible that a call fails, but if it is false, calling object will never succeed. Note that classes are callable (calling a class returns a new instance); class instances are callable if they have a __call__() method.

chr(i)
: Return a string of one character whose ASCII code is the integer i. For example, chr(97) returns the string 'a'. This is the inverse of ord(). The argument must be in the range [0..255], inclusive; ValueError will be raised if i is outside that range. See also unichr().

classmethod(function)
Return a class method for function.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C(object):
    @classmethod
    def f(cls, arg1, arg2, ...):
        ...
```

The @classmethod form is a function decorator – see the description of function definitions in function for details.

It can be called either on the class (such as C.f()) or on an instance (such as C().f()). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see staticmethod() in this section.

For more information on class methods, consult the documentation on the standard type hierarchy in types.

New in version 2.2.

Changed in version 2.4: Function decorator syntax added.

cmp(x, y)
Compare the two objects x and y and return an integer according to the outcome. The return value is negative if x < y, zero if x == y and strictly positive if x > y.

compile(source, filename, mode[, flags[, dont_inherit]])
Compile the source into a code or AST object. Code objects can be executed by an exec statement or evaluated by a call to eval(). source can either be a Unicode string, a Latin-1 encoded string or an AST object. Refer to the ast module documentation for information on how to work with AST objects.

The filename argument should give the file from which the code was read; pass some recognizable value if it wasn't read from a file ('<string>' is commonly used).

The mode argument specifies what kind of code must be compiled; it can be 'exec' if source consists of a sequence of statements, 'eval' if it consists of a single expression, or 'single' if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something other than None will be printed).

The optional arguments flags and dont_inherit control which future statements (see PEP 236) affect the compilation of source. If neither is present (or both are zero) the code is compiled with those future statements that are in effect in the code that is calling compile(). If the flags argument is given and dont_inherit is not (or is zero) then the future statements specified by the flags argument are used in addition to those that would be used anyway. If dont_inherit is a non-zero integer then the flags argument is it – the future statements in effect around the call to compile are ignored.

Future statements are specified by bits which can be bitwise ORed together to specify multiple statements. The bitfield required to specify a given feature can be found as the compiler_flag attribute on the _Feature instance in the __future__ module.

This function raises SyntaxError if the compiled source is invalid, and TypeError if the source contains null bytes.

If you want to parse Python code into its AST representation, see ast.parse().

Note: When compiling a string with multi-line code in 'single' or 'eval' mode, input must be terminated by at least one newline character. This is to facilitate detection of incomplete and complete statements in the code module.

Changed in version 2.3: The flags and dont_inherit arguments were added.

Changed in version 2.6: Support for compiling AST objects.

Changed in version 2.7: Allowed use of Windows and Mac newlines. Also input in 'exec' mode does not have to end in a newline anymore.

class complex([real[, imag]])
Return a complex number with the value real + imag*1j or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If imag is omitted, it defaults to zero and the function serves as a numeric conversion function like int(), long() and float(). If both arguments are omitted, returns 0j.

Note: When converting from a string, the string must not contain whitespace around the central + or - operator. For example, complex('1+2j') is fine, but complex('1 + 2j') raises ValueError.

The complex type is described in Numeric Types — int, float, long, complex.

delattr(object, name)
This is a relative of setattr(). The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, delattr(x, 'foobar') is equivalent to del x.foobar.

class dict(**kwarg)
class dict(mapping, **kwarg)
class dict(iterable, **kwarg)
Create a new dictionary. The dict object is the dictionary class. See dict and Mapping Types — dict for documentation about this class.

For other containers see the built-in list, set, and tuple classes, as well as the collections module.

dir([object])
Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

If the object has a method named __dir__(), this method will be called and must return the list of attributes. This allows objects that implement a custom __getattr__() or __getattribute__() function to customize the way dir() reports their attributes.

If the object does not provide __dir__(), the function tries its best to gather information from the object's __dict__ attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom __getattr__().

The default dir() mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module's attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.

- Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes.

The resulting list is sorted alphabetically. For example:

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__doc__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape(object):
        def __dir__(self):
            return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
['area', 'perimeter', 'location']
```

Note: Because dir() is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

---

divmod(a, b)

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using long division. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as (a // b, a % b). For floating point numbers the result is (q, a % b), where q is usually math.floor(a / b) but may be 1 less than that. In any case q * b + a % b is very close to a, if a % b is non-zero it has the same sign as b, and 0 <= abs(a % b) < abs(b).

Changed in version 2.3: Using divmod() with complex numbers is deprecated.

enumerate(sequence, start=0)

Return an enumerate object. sequence must be a sequence, an iterator, or some other object which supports iteration. The next() method of the iterator returned by enumerate() returns a tuple containing a count (from start which defaults to 0) and the values obtained from iterating over sequence:

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

New in version 2.3.

Changed in version 2.6: The start parameter was added.

eval(expression[, globals[, locals]])

The arguments are a Unicode or Latin-1 encoded string and optional globals and locals. If provided, globals must be a dictionary. If provided, locals can be any mapping object.

Changed in version 2.4: formerly locals was required to be a dictionary.

The expression argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the globals and locals dictionaries as global and local namespace. If the globals dictionary is present and lacks '__builtins__', the current globals are copied into globals before expression is parsed. This means that expression normally has full access to the standard __builtin__ module and restricted environments are propagated. If the locals dictionary is omitted it defaults to the globals dictionary. If both dictionaries are omitted, the expression is executed in the environment where eval() is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```
>>> x = 1
>>> print eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by compile()). In this case pass a code object instead of a string. If the code object has been compiled with 'exec' as the mode argument, eval()'s return value will be None.

Hints: dynamic execution of statements is supported by the exec statement. Execution of statements from a file is supported by the execfile() function. The globals() and locals() functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by eval() or execfile().

See ast.literal_eval() for a function that can safely evaluate strings with expressions containing only literals.

execfile(filename[, globals[, locals]])

This function is similar to the exec statement, but parses a file instead of a string. It is different from the import statement in that it does not use the module administration — it reads the file unconditionally and does not create a new module.[1]

The arguments are a file name and two optional dictionaries. The file is parsed and evaluated as a sequence of Python statements (similarly to a module) using the globals and locals dictionaries as global and local namespace. If provided, locals can be any mapping object. Remember that at module level, globals and locals are the same dictionary. If two separate objects are passed as globals and locals, the code will be executed as if it were embedded in a class definition.

Changed in version 2.4: formerly locals was required to be a dictionary.

If the locals dictionary is omitted it defaults to the globals dictionary. If both dictionaries are omitted, the expression is executed in the environment where execfile() is called. The return value is None.

Note: The default locals act as described for function locals() below: modifications to the default locals dictionary should not be attempted. Pass an explicit locals dictionary if you need to see effects of the code on locals after function execfile() returns. execfile() cannot be used reliably to modify a function's locals.

---

file(name[, mode[, buffering]])

Constructor function for the file type, described further in section File Objects. The constructor's arguments are the same as those of the open() built-in function described below.

[1] It is used relatively rarely so does not warrant being made into a statement.

When opening a file, it's preferable to use open() instead of invoking this constructor directly. file is more suited to type testing (for example, writing isinstance(f, file)).

New in version 2.2.

filter(function, iterable)
Construct a list from those elements of iterable for which function returns true. iterable may be either a sequence, a container which supports iteration, or an iterator. If iterable is a string or a tuple, the result also has that type; otherwise it is always a list. If function is None, the identity function is assumed, that is, all elements of iterable that are false are removed.

Note that filter(function, iterable) is equivalent to [item for item in iterable if function(item)] if function is not None and [item for item in iterable if item] if function is None.

See itertools.ifilter() and itertools.ifilterfalse() for iterator versions of this function, including a variation that filters for elements where the function returns false.

class float([x])
Return a floating point number constructed from a number or string x.

If the argument is a string, it must contain a possibly signed decimal or floating point number, possibly embedded in whitespace. The argument may also be [+|-]nan or [+|-]inf. Otherwise, the argument may be a plain or long integer or a floating point number, and a floating point number with the same value (within Python's floating point precision) is returned. If no argument is given, returns 0.0.

Note: When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. Float accepts the strings nan, inf and -inf for NaN and positive or negative infinity. The case and a leading + are ignored as well as a leading - is ignored for NaN. Float always represents NaN and infinity as nan, inf or -inf.

The float type is described in Numeric Types — int, float, long, complex.

format(value[, format_spec])
Convert a value to a "formatted" representation, as controlled by format_spec. The interpretation of format_spec will depend on the type of the value argument, however there is a standard formatting syntax that is used by most built-in types: Format Specification Mini-Language.

Note: format(value, format_spec) merely calls value.__format__(format_spec).

New in version 2.6.

class frozenset([iterable])
Return a new frozenset object, optionally with elements taken from iterable. frozenset is a built-in class. See frozenset and Set Types — set, frozenset for documentation about this class.

For other containers see the built-in set, list, tuple, and dict classes, as well as the collections module.

New in version 2.4.

getattr(object, name[, default])
Return the value of the named attribute of object. name must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, getattr(x, 'foobar') is equivalent to x.foobar. If the named attribute does not exist, default is returned if provided, otherwise AttributeError is raised.

globals()
Return a dictionary representing the current global symbol table. This is always the dictionary of the

current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

hasattr(object, name)
: The arguments are an object and a string. The result is True if the string is the name of one of the object's attributes, False if not. (This is implemented by calling getattr(object, name) and seeing whether it raises an exception or not.)

hash(object)
: Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

help([object])
: Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.

  This function is added to the built-in namespace by the site module.

  New in version 2.2.

hex(x)
: Convert an integer number (of any size) to a lowercase hexadecimal string prefixed with "0x", for example:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
>>> hex(1L)
'0x1L'
```

  If x is not a Python int or long object, it has to define a __hex__() method that returns a string.

  See also int() for converting a hexadecimal string to an integer using a base of 16.

  Note: To obtain a hexadecimal string representation for a float, use the float.hex() method.

  Changed in version 2.4: Formerly only returned an unsigned literal.

id(object)
: Return the "identity" of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id() value.

  CPython implementation detail: This is the address of the object in memory.

input([prompt])
: Equivalent to eval(raw_input(prompt)).

  This function does not catch user errors. If the input is not syntactically valid, a SyntaxError will be raised. Other exceptions may be raised if there is an error during evaluation.

  If the readline module was loaded, then input() will use it to provide elaborate line editing and history features.

  Consider using the raw_input() function for general input from users.

class int(x=0)
class int(x, base=10)
Return an integer object constructed from a number or string x, or return 0 if no arguments are given. If x is a number, it can be a plain integer, a long integer, or a floating point number. If x is floating point, the conversion truncates towards zero. If the argument is outside the integer range, the function returns a long object instead.

If x is not a number or if base is given, then x must be a string or Unicode object representing an integer literal in radix base. Optionally, the literal can be preceded by + or - (with no space in between) and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with a to z (or A to Z) having values 10 to 35. The default base is 10. The allowed values are 0 and 2–36. Base-2, -8, and -16 literals can be optionally prefixed with 0b/0B, 0o/0O/0, or 0x/0X, as with integer literals in code. Base 0 means to interpret the string exactly as an integer literal, so that the actual base is 2, 8, 10, or 16.

The integer type is described in Numeric Types — int, float, long, complex.

isinstance(object, classinfo)
Return true if the object argument is an instance of the classinfo argument, or of a (direct, indirect or virtual) subclass thereof. Also return true if classinfo is a type object (new-style class) and object is an object of that type or of a (direct, indirect or virtual) subclass thereof. If object is not a class instance or an object of the given type, the function always returns false. If classinfo is a tuple of class or type objects (or recursively, other such tuples), return true if object is an instance of any of the classes or types. If classinfo is not a class, type, or tuple of classes, types, and such tuples, a TypeError exception is raised.

Changed in version 2.2: Support for a tuple of type information was added.

issubclass(class, classinfo)
Return true if class is a subclass (direct, indirect or virtual) of classinfo. A class is considered a subclass of itself. classinfo may be a tuple of class objects, in which case every entry in classinfo will be checked. In any other case, a TypeError exception is raised.

Changed in version 2.3: Support for a tuple of type information was added.

iter(o[, sentinel])
Return an iterator object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, o must be a collection object which supports the iteration protocol (the __iter__() method), or it must support the sequence protocol (the __getitem__() method with integer arguments starting at 0). If it does not support either of those protocols, TypeError is raised. If the second argument, sentinel, is given, then o must be a callable object. The iterator created in this case will call o with no arguments for each call to its next() method; if the value returned is equal to sentinel, StopIteration will be raised, otherwise the value will be returned.

One useful application of the second form of iter() is to read lines of a file until a certain line is reached. The following example reads a file until the readline() method returns an empty string:

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)
```

New in version 2.2.

len(s)
Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

class list([iterable])
: Return a list whose items are the same and in the same order as iterable's items. iterable may be either a sequence, a container that supports iteration, or an iterator object. If iterable is already a list, a copy is made and returned, similar to iterable[:]. For instance, list('abc') returns ['a', 'b', 'c'] and list((1, 2, 3)) returns [1, 2, 3]. If no argument is given, returns a new empty list, [].

  list is a mutable sequence type, as documented in Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange. For other containers see the built in dict, set, and tuple classes, and the collections module.

locals()
: Update and return a dictionary representing the current local symbol table. Free variables are returned by locals() when it is called in function blocks, but not in class blocks.

  Note: The contents of this dictionary should not be modified; changes may not affect the values of local and free variables used by the interpreter.

  ---

class long(x=0)
class long(x, base=10)
: Return a long integer object constructed from a string or number x. If the argument is a string, it must contain a possibly signed number of arbitrary size, possibly embedded in whitespace. The base argument is interpreted in the same way as for int(), and may only be given when x is a string. Otherwise, the argument may be a plain or long integer or a floating point number, and a long integer with the same value is returned. Conversion of floating point numbers to integers truncates (towards zero). If no arguments are given, returns 0L.

  The long type is described in Numeric Types — int, float, long, complex.

map(function, iterable, ...)
: Apply function to every item of iterable and return a list of the results. If additional iterable arguments are passed, function must take that many arguments and is applied to the items from all iterables in parallel. If one iterable is shorter than another it is assumed to be extended with None items. If function is None, the identity function is assumed; if there are multiple arguments, map() returns a list consisting of tuples containing the corresponding items from all iterables (a kind of transpose operation). The iterable arguments may be a sequence or any iterable object; the result is always a list.

max(iterable[, key])
max(arg1, arg2, *args[, key])
: Return the largest item in an iterable or the largest of two or more arguments.

  If one positional argument is provided, iterable must be a non-empty iterable (such as a non-empty string, tuple or list). The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

  The optional key argument specifies a one-argument ordering function like that used for list.sort(). The key argument, if supplied, must be in keyword form (for example, max(a,b,c,key=func)).

  Changed in version 2.5: Added support for the optional key argument.

memoryview(obj)
: Return a "memory view" object created from the given argument. See memoryview type for more information.

min(iterable[, key])
min(arg1, arg2, *args[, key])
: Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, iterable must be a non-empty iterable (such as a non-empty string, tuple or list). The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

The optional key argument specifies a one-argument ordering function like that used for list.sort(). The key argument, if supplied, must be in keyword form (for example, min(a,b,c,key=func)).

Changed in version 2.5: Added support for the optional key argument.

next(iterator[, default])
Retrieve the next item from the iterator by calling its next() method. If default is given, it is returned if the iterator is exhausted, otherwise StopIteration is raised.

New in version 2.6.

class object
Return a new featureless object. object is a base for all new style classes. It has the methods that are common to all instances of new style classes.

New in version 2.2.

Changed in version 2.3: This function does not accept any arguments. Formerly, it accepted arguments but ignored them.

oct(x)
Convert an integer number (of any size) to an octal string. The result is a valid Python expression.

Changed in version 2.4: Formerly only returned an unsigned literal.

open(name[, mode[, buffering]])
Open a file, returning an object of the file type described in section File Objects. If the file cannot be opened, IOError is raised. When opening a file, it's preferable to use open() instead of invoking the file constructor directly.

The first two arguments are the same as for stdio's fopen(): name is the file name to be opened, and mode is a string indicating how the file is to be opened.

The most commonly-used values of mode are `'r'` for reading, `'w'` for writing (truncating the file if it already exists), and `'a'` for appending (which on some Unix systems means that all writes append to the end of the file regardless of the current seek position). If mode is omitted, it defaults to `'r'`. The default is to use text mode, which may convert `'\n'` characters to a platform-specific representation on writing and back on reading. Thus, when opening a binary file, you should append `'b'` to the mode value to open the file in binary mode, which will improve portability. (Appending `'b'` is useful even on systems that don't treat binary and text files differently, where it serves as documentation.) See below for more possible values of mode.

The optional buffering argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size (in bytes). A negative buffering means to use the system default, which is usually line buffered for tty devices and fully buffered for other files. If omitted, the system default is used.[2]

Modes `'r+'`, `'w+'` and `'a+'` open the file for updating (reading and writing); note that `'w+'` truncates the file. Append `'b'` to the mode to open the file in binary mode, on systems that differentiate between binary and text files; on systems that don't have this distinction, adding the `'b'` has no effect.

In addition to the standard fopen() values mode may be `'U'` or `'rU'`. Python is usually built with universal newlines support; supplying `'U'` opens the file as a text file, but lines may be terminated by any of the following: the Unix end-of-line convention `'\n'`, the Macintosh convention `'\r'`, or the

[2] Specifying a buffer size currently has no effect on systems that don't have setvbuf(). The interface to specify the buffer size is not done using a method that calls setvbuf(), because that may dump core when called after any I/O has been performed, and there's no reliable way to determine whether this is the case.

Windows convention `'\r\n'`. All of these external representations are seen as `'\n'` by the Python program. If Python is built without universal newlines support a mode with `'U'` is the same as normal text mode. Note that file objects so opened also have an attribute called newlines which has a value of None (if no newlines have yet been seen), `'\n'`, `'\r'`, `'\r\n'`, or a tuple containing all the newline types seen.

Python enforces that the mode, after stripping `'U'`, begins with `'r'`, `'w'` or `'a'`.

Python provides many file handling modules including fileinput, os, os.path, tempfile, and shutil.

Changed in version 2.5: Restriction on first letter of mode string introduced.

ord(c)
Given a string of length one, return an integer representing the Unicode code point of the character when the argument is a unicode object, or the value of the byte when the argument is an 8-bit string. For example, ord('a') returns the integer 97, ord(u'\u2020') returns 8224. This is the inverse of chr() for 8-bit strings and of unichr() for unicode objects. If a unicode argument is given and Python was built with UCS2 Unicode, then the character's code point must be in the range [0..65535] inclusive; otherwise the string length is two, and a TypeError will be raised.

pow(x, y[, z])
Return x to the power y; if z is present, return x to the power y, modulo z (computed more efficiently than pow(x, y) % z). The two-argument form pow(x, y) is equivalent to using the power operator: x**y.

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For int and long int operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, 10**2 returns 100, but 10**-2 returns 0.01. (This last feature was added in Python 2.2. In Python 2.1 and before, if both arguments were of integer types and the second argument was negative, an exception was raised.) If the second argument is negative, the third argument must be omitted. If z is present, x and y must be of integer types, and y must be non-negative. (This restriction was added in Python 2.2. In Python 2.1 and before, floating 3-argument pow() returned platform-dependent results depending on floating-point rounding accidents.)

print(*objects, sep=' ', end='\n', file=sys.stdout)
Print objects to the stream file, separated by sep and followed by end. sep, end and file, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like str() does and written to the stream, separated by sep and followed by end. Both sep and end must be strings; they can also be None, which means to use the default values. If no objects are given, print() will just write end.

The file argument must be an object with a write(string) method; if it is not present or None, sys.stdout will be used. Output buffering is determined by file. Use file.flush() to ensure, for instance, immediate appearance on a screen.

Note: This function is not normally available as a built-in since the name print is recognized as the print statement. To disable the statement and use the print() function, use this future statement at the top of your module:

```
from __future__ import print_function
```

New in version 2.6.

class property([fget[, fset[, fdel[, doc]]]])
Return a property attribute for new-style classes (classes that derive from object).

fget is a function for getting an attribute value. fset is a function for setting an attribute value. fdel is a function for deleting an attribute value. And doc creates a docstring for the attribute.

A typical use is to define a managed attribute x:

```
class C(object):
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

If c is an instance of C, c.x will invoke the getter, c.x = value will invoke the setter and del c.x the deleter.

If given, doc will be the docstring of the property attribute. Otherwise, the property will copy fget's docstring (if it exists). This makes it possible to create read-only properties easily using property() as a decorator:

```
class Parrot(object):
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

The @property decorator turns the voltage() method into a "getter" for a read-only attribute with the same name, and it sets the docstring for voltage to "Get the current voltage."

A property object has getter, setter, and deleter methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```
class C(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (x in this case.)

The returned property object also has the attributes fget, fset, and fdel corresponding to the constructor arguments.

New in version 2.2.

Changed in version 2.5: Use fget's docstring if no doc given.

Changed in version 2.6: The getter, setter, and deleter attributes were added.

range(stop)
range(start, stop[, step])
This is a versatile function to create lists containing arithmetic progressions. It is most often used in for loops. The arguments must be plain integers. If the step argument is omitted, it defaults to 1. If the start argument is omitted, it defaults to 0. The full form returns a list of plain integers [start, start + step, start + 2 * step, ...]. If step is positive, the last element is the largest start + i * step less than stop; if step is negative, the last element is the smallest start + i * step greater than stop. step must not be zero (or else ValueError is raised). Example:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

raw_input([prompt])
If the prompt argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, EOFError is raised. Example:

```
>>> s = raw_input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the readline module was loaded, then raw_input() will use it to provide elaborate line editing and history features.

reduce(function, iterable[, initializer])
Apply function of two arguments cumulatively to the items of iterable, from left to right, so as to reduce the iterable to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). The left argument, x, is the accumulated value and the right argument, y, is the update value from the iterable. If the optional initializer is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty. If initializer is not given and iterable contains only one item, the first item is returned. Roughly equivalent to:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        try:
            initializer = next(it)
        except StopIteration:
            raise TypeError('reduce() of empty sequence with no initial value')
    accum_value = initializer
    for x in it:
        accum_value = function(accum_value, x)
    return accum_value
```

reload(module)

Reload a previously imported module. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the module argument).

When reload(module) is executed:

- Python modules' code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module's dictionary. The init function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects — with a try statement it can test for the table's presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is generally not very useful to reload built-in or dynamically loaded modules. Reloading sys, __main__, builtins and other key modules is not recommended. In many cases extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using from ... import ..., calling reload() for the other module does not redefine the objects imported from it — one way around this is to re-execute the from statement, another is to use import and qualified names (module.*name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

repr(object)

Return a string containing a printable representation of an object. This is the same value yielded by

conversions (reverse quotes). It is sometimes useful to be able to access this operation as an ordinary function. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval(), otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a __repr__() method.

reversed(seq)
Return a reverse iterator. seq must be an object which has a __reversed__() method or supports the sequence protocol (the __len__() method and the __getitem__() method with integer arguments starting at 0).

New in version 2.4.

Changed in version 2.6: Added the possibility to write a custom __reversed__() method.

round(number[, ndigits])
Return the floating point value number rounded to ndigits digits after the decimal point. If ndigits is omitted, it defaults to zero. The result is a floating point number. Values are rounded to the closest multiple of 10 to the power minus ndigits; if two multiples are equally close, rounding is done away from 0 (so, for example, round(0.5) is 1.0 and round(-0.5) is -1.0).

Note: The behavior of round() for floats can be surprising: for example, round(2.675, 2) gives 2.67 instead of the expected 2.68. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See tut-fp-issues for more information.

---

class set([iterable])
Return a new set object, optionally with elements taken from iterable. set is a built-in class. See set and Set Types — set, frozenset for documentation about this class.

For other containers see the built-in frozenset, list, tuple, and dict classes, as well as the collections module.

New in version 2.4.

setattr(object, name, value)
This is the counterpart of getattr(). The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, setattr(x, 'foobar', 123) is equivalent to x.foobar = 123.

class slice(stop)
class slice(start, stop[, step])
Return a slice object representing the set of indices specified by range(start, stop, step). The start and step arguments default to None. Slice objects have read-only data attributes start, stop and step which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: a[start:stop:step] or a[start:stop, i]. See itertools.islice() for an alternate version that returns an iterator.

sorted(iterable[, cmp[, key[, reverse]]])
Return a new sorted list from the items in iterable.

The optional arguments cmp, key, and reverse have the same meaning as those for the list.sort() method (described in section Mutable Sequence Types).

cmp specifies a custom comparison function of two arguments (iterable elements) which should return a negative, zero or positive number depending on whether the first argument is considered smaller

than, equal to, or larger than the second argument: cmp=lambda x,y: cmp(x.lower(), y.lower()). The default value is None.

key specifies a function of one argument that is used to extract a comparison key from each list element: key=str.lower. The default value is None (compare the elements directly).

reverse is a boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

In general, the key and reverse conversion processes are much faster than specifying an equivalent cmp function. This is because cmp is called multiple times for each list element while key and reverse touch each element only once. Use functools.cmp_to_key() to convert an old-style cmp function to a key function.

The built-in sorted() function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see sortinghowto.

New in version 2.4.

staticmethod(function)
Return a static method for function.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class C(object):
    @staticmethod
    def f(arg1, arg2, ...):
        ...
```

The @staticmethod form is a function decorator – see the description of function definitions in function for details.

It can be called either on the class (such as C.f()) or on an instance (such as C().f()). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. Also see classmethod() for a variant that is useful for creating alternate class constructors.

For more information on static methods, consult the documentation on the standard type hierarchy in types.

New in version 2.2.

Changed in version 2.4: Function decorator syntax added.

class str(object='')
Return a string containing a nicely printable representation of an object. For strings, this returns the string itself. The difference with repr(object) is that str(object) does not always attempt to return a string that is acceptable to eval(); its goal is to return a printable string. If no argument is given, returns the empty string, ''.

For more information on strings see Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange which describes sequence functionality (strings are sequences), and also the string-specific methods described in the String Methods section. To output formatted strings use template strings or the % operator described in the String Formatting Operations section. In addition see the String Services section. See also unicode().

sum(iterable[, start])
Sums start and the items of an iterable from left to right and returns the total. start defaults to 0. The iterable's items are normally numbers, and the start value is not allowed to be a string.

For some use cases, there are good alternatives to sum(). The preferred, fast way to concatenate a sequence of strings is by calling ''.join(sequence). To add floating point values with extended precision, see math.fsum(). To concatenate a series of iterables, consider using itertools.chain().

New in version 2.3.

super(type[, object-or-type])
Return a proxy object that delegates method calls to a parent or sibling class of type. This is useful for accessing inherited methods that have been overridden in a class. The search order is same as that used by getattr() except that the type itself is skipped.

The __mro__ attribute of the type lists the method resolution search order used by both getattr() and super(). The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, isinstance(obj, type) must be true. If the second argument is a type, issubclass(type2, type) must be true (this is useful for classmethods).

Note: super() only works for new-style classes.

There are two typical use cases for super. In a class hierarchy with single inheritance, super can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of super in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement "diamond diagrams" where multiple base classes implement the same method. Good design dictates that this method have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```
class C(B):
    def method(self, arg):
        super(C, self).method(arg)
```

Note that super() is implemented as part of the binding process for explicit dotted attribute lookups such as super().__getitem__(name). It does so by implementing its own __getattribute__() method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly, super() is undefined for implicit lookups using statements or operators such as super()[name].

Also note that super() is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references.

For practical suggestions on how to design cooperative classes using super(), see guide to using super().

New in version 2.2.

tuple([iterable])
Return a tuple whose items are the same and in the same order as iterable's items. iterable may be a sequence, a container that supports iteration, or an iterator object. If iterable is already a tuple, it is returned unchanged. For instance, tuple('abc') returns ('a', 'b', 'c') and tuple([1, 2, 3]) returns (1, 2, 3). If no argument is given, returns a new empty tuple, ().

tuple is an immutable sequence type, as documented in Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange. For other containers see the built in dict, list, and set classes, and the collections module.

class type(object)
class type(name, bases, dict)
With one argument, return the type of an object. The return value is a type object. The isinstance() built-in function is recommended for testing the type of an object.

With three arguments, return a new type object. This is essentially a dynamic form of the class statement. The name string is the class name and becomes the __name__ attribute; the bases tuple itemizes the base classes and becomes the __bases__ attribute; and the dict dictionary is the namespace containing definitions for class body and becomes the __dict__ attribute. For example, the following two statements create identical type objects:

```
>>> class X(object):
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

New in version 2.2.

unichr(i)
Return the Unicode string of one character whose Unicode code is the integer i. For example, unichr(97) returns the string u'a'. This is the inverse of ord() for Unicode strings. The valid range for the argument depends how Python was configured – it may be either UCS2 [0..0xFFFF] or UCS4 [0..0x10FFFF]. ValueError is raised otherwise. For ASCII and 8-bit strings see chr().

New in version 2.0.

unicode(object='')
unicode(object[, encoding[, errors]])
Return the Unicode string version of object using one of the following modes:

If encoding and/or errors are given, unicode() will decode the object which can either be an 8-bit string or a character buffer using the codec for encoding. The encoding parameter is a string giving the name of an encoding; if the encoding is not known, LookupError is raised. Error handling is done according to errors; this specifies the treatment of characters which are invalid in the input encoding. If errors is 'strict' (the default), a ValueError is raised on errors, while a value of 'ignore' causes errors to be silently ignored, and a value of 'replace' causes the official Unicode replacement character, U+FFFD, to be used to replace input characters which cannot be decoded. See also the codecs module.

If no optional parameters are given, unicode() will mimic the behaviour of str() except that it returns Unicode strings instead of 8-bit strings. More precisely, if object is a Unicode string or subclass it will return that Unicode string without any additional decoding applied.

For objects which provide a __unicode__() method, it will call this method without arguments to create a Unicode string. For all other objects, the 8-bit string version or representation is requested and then converted to a Unicode string using the codec for the default encoding in 'strict' mode.

For more information on Unicode strings see Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange which describes sequence functionality (Unicode strings are sequences), and also the string-specific methods described in the String Methods section. To output formatted strings use template strings or the % operator described in the String Formatting Operations section. In addition see the String Services section. See also str().

New in version 2.0.

Changed in version 2.2: Support for __unicode__() added.

vars([object])
: Return the __dict__ attribute for a module, class, instance, or any other object with a __dict__ attribute.

  Objects such as modules and instances have an updateable __dict__ attribute; however, other objects may have write restrictions on their __dict__ attributes (for example, new-style classes use a dictproxy to prevent direct dictionary updates).

  Without an argument, vars() acts like locals(). Note, the locals dictionary is only useful for reads since updates to the locals dictionary are ignored.

xrange(stop)
xrange(start, stop[, step])
: This function is very similar to range(), but returns an xrange object instead of a list. This is an opaque sequence type which yields the same values as the corresponding list, without actually storing them all simultaneously. The advantage of xrange() over range() is minimal (since xrange() still has to create the values when asked for them) except when a very large range is used on a memory-starved machine or when all of the range's elements are never used (such as when the loop is usually terminated with break). For more information on xrange objects, see XRange Type and Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange.

  CPython implementation detail: xrange() is intended to be simple and fast. Implementations may impose restrictions to achieve this. The C implementation of Python restricts all arguments to native C longs ("short" Python integers), and also requires that the number of elements fit in a native C long. If a larger range is needed, an alternate version can be crafted using the itertools module: islice(count(start, step), (stop-start+step-1+2*(step<0))//step).

zip([iterable, ...])
: This function returns a list of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The returned list is truncated in length to the length of the shortest argument sequence. When there are multiple arguments which are all of the same length, zip() is similar to map() with an initial argument of None. With a single sequence argument, it returns a list of 1-tuples. With no arguments, it returns an empty list.

  The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using zip(*[iter(s)]*n).

  zip() in conjunction with the * operator can be used to unzip a list:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> zipped
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zipped)
>>> x == list(x2) and y == list(y2)
True
```

  New in version 2.0.

  Changed in version 2.4: Formerly, zip() required at least one argument and zip() raised a TypeError instead of returning an empty list.

__import__(name[, globals[, locals[, fromlist[, level]]]])

Note: This is an advanced function that is not needed in everyday Python programming, unlike

importlib.import_module().

This function is invoked by the import statement. It can be replaced (by importing the __builtin__ module and assigning to __builtin__.__import__) in order to change semantics of the import statement, but nowadays it is usually simpler to use import hooks (see PEP 302). Direct use of __import__() is rare, except in cases where you want to import a module whose name is only known at runtime.

The function imports the module name, potentially using the given globals and locals to determine how to interpret the name in a package context. The fromlist gives the names of objects or submodules that should be imported from the module given by name. The standard implementation does not use its locals argument at all, and uses its globals only to determine the package context of the import statement.

level specifies whether to use absolute or relative imports. The default is -1 which indicates both absolute and relative imports will be attempted. 0 means only perform absolute imports. Positive values for level indicate the number of parent directories to search relative to the directory of the module calling __import__().

When the name variable is of the form package.module, normally, the top-level package (the name up till the first dot) is returned, not the module named by name. However, when a non-empty fromlist argument is given, the module named by name is returned.

For example, the statement import spam results in bytecode resembling the following code:

```
spam = __import__('spam', globals(), locals(), [], -1)
```

The statement import spam.ham results in this call:

```
spam = __import__('spam.ham', globals(), locals(), [], -1)
```

Note how __import__() returns the toplevel module here because this is the object that is bound to a name by the import statement.

On the other hand, the statement from spam.ham import eggs, sausage as saus results in

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], -1)
eggs = _temp.eggs
saus = _temp.sausage
```

Here, the spam.ham module is returned from __import__(). From this object, the names to import are retrieved and assigned to their respective names.

If you simply want to import a module (potentially within a package) by name, use importlib.import_module().

Changed in version 2.5: The level parameter was added.

Changed in version 2.5: Keyword support for parameters was added.

CHAPTER

THREE

NON-ESSENTIAL BUILT-IN FUNCTIONS

There are several built-in functions that are no longer essential to learn, know or use in modern Python programming. They have been kept here to maintain backwards compatibility with programs written for older versions of Python.

Python programmers, trainers, students and book writers should feel free to bypass these functions without concerns about missing something important.

apply(function, args[, keywords])
: The function argument must be a callable object (a user-defined or built-in function or method, or a class object) and the args argument must be a sequence. The function is called with args as the argument list; the number of arguments is the length of the tuple. If the optional keywords argument is present, it must be a dictionary whose keys are strings. It specifies keyword arguments to be added to the end of the argument list. Calling apply() is different from just calling function(args), since in that case there is always exactly one argument. The use of apply() is equivalent to function(*args, **keywords).

  Deprecated since version 2.3: Use function(*args, **keywords) instead of apply(function, args, keywords) (see tut-unpacking-arguments).

buffer(object[, offset[, size]])
: The object argument must be an object that supports the buffer call interface (such as strings, arrays, and buffers). A new buffer object will be created which references the object argument. The buffer object will be a slice from the beginning of object (or from the specified offset). The slice will extend to the end of object (or will have a length given by the size argument).

coerce(x, y)
: Return a tuple consisting of the two numeric arguments converted to a common type, using the same rules as used by arithmetic operations. If coercion is not possible, raise TypeError.

intern(string)
: Enter string in the table of "interned" strings and return the interned string – which is string itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

  Changed in version 2.3: Interned strings are not immortal (like they used to be in Python 2.2 and before); you must keep a reference to the return value of intern() around to benefit from it.

CHAPTER

FOUR

BUILT-IN CONSTANTS

A small number of constants live in the built-in namespace. They are:

False
: The false value of the bool type.

  New in version 2.3.

True
: The true value of the bool type.

  New in version 2.3.

None
: The sole value of types.NoneType. None is frequently used to represent the absence of a value, as when default arguments are not passed to a function.

  Changed in version 2.4: Assignments to None are illegal and raise a SyntaxError.

NotImplemented
: Special value which can be returned by the "rich comparison" special methods (__eq__(), __lt__(), and friends), to indicate that the comparison is not implemented with respect to the other type.

Ellipsis
: Special value used in conjunction with extended slicing syntax.

__debug__
: This constant is true if Python was not started with an -O option. See also the assert statement.

Note: The names None and __debug__ cannot be reassigned (assignments to them, even as an attribute name, raise SyntaxError), so they can be considered "true" constants.

Changed in version 2.7: Assignments to __debug__ as an attribute became illegal.

4.1 Constants added by the site module

The site module (which is imported automatically during startup, except if the -S command-line option is given) adds several constants to the built-in namespace. They are useful for the interactive interpreter shell and should not be used in programs.

quit([code=None])
exit([code=None])
: Objects that when printed, print a message like "Use quit() or Ctrl-D (i.e. EOF) to exit", and when called, raise SystemExit with the specified exit code.

copyright
credits
: Objects that when printed or called, print the text of copyright or credits, respectively.

license
: Object that when printed, prints the message "Type license() to see the full license text", and when called, displays the full license text in a pager-like fashion (one screen at a time).

CHAPTER

FIVE

BUILT-IN TYPES

The following sections describe the standard types that are built into the interpreter.

Note: Historically (until release 2.2), Python's built-in types have differed from user-defined types because it was not possible to use the built-in types as the basis for object-oriented inheritance. This limitation no longer exists.

The principal built-in types are numerics, sequences, mappings, files, classes, instances and exceptions.

Some operations are supported by several object types; in particular, practically all objects can be compared, tested for truth value, and converted to a string (with the repr() function or the slightly different str() function). The latter function is implicitly used when an object is written by the print() function.

5.1 Truth Value Testing

Any object can be tested for truth value, for use in an if or while condition or as operand of the Boolean operations below. The following values are considered false:

- None
- False
- zero of any numeric type, for example, 0, 0L, 0.0, 0j.
- any empty sequence, for example, '', (), [].
- any empty mapping, for example, {}.
- instances of user-defined classes, if the class defines a __nonzero__() or __len__() method, when that method returns the integer zero or bool value False.[1]

All other values are considered true — so objects of many types are always true.

Operations and built-in functions that have a Boolean result always return 0 or False for false and 1 or True for true, unless otherwise stated. (Important exception: the Boolean operations or and and always return one of their operands.)

[1] Additional information on these special methods may be found in the Python Reference Manual (customization).

5.2 Boolean Operations — and, or, not

These are the Boolean operations, ordered by ascending priority:

Operation	Result	Notes
x or y	if x is false, then y, else x	(1)
x and y	if x is false, then x, else y	(2)
not x	if x is false, then True, else False	(3)

Notes:

(1) This is a short-circuit operator, so it only evaluates the second argument if the first one is false.

(2) This is a short-circuit operator, so it only evaluates the second argument if the first one is true.

(3) not has a lower priority than non-Boolean operators, so not a == b is interpreted as not (a == b), and a == not b is a syntax error.

5.3 Comparisons

Comparison operations are supported by all objects. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, x < y <= z is equivalent to x < y and y <= z, except that y is evaluated only once (but in both cases z is not evaluated at all when x < y is found to be false).

This table summarizes the comparison operations:

Operation	Meaning	Notes
<	strictly less than	
<=	less than or equal	
>	strictly greater than	
>=	greater than or equal	
==	equal	
!=	not equal	(1)
is	object identity	
is not	negated object identity	

Notes:

(1) != can also be written <>, but this is an obsolete usage kept for backwards compatibility only. New code should always use !=.

Objects of different types, except different numeric types and different string types, never compare equal; such objects are ordered consistently but arbitrarily (so that sorting a heterogeneous array yields a consistent result). Furthermore, some types (for example, file objects) support only a degenerate notion of comparison where any two objects of that type are unequal. Again, such objects are ordered arbitrarily but consistently. The <, <=, > and >= operators will raise a TypeError exception when any operand is a complex number.

Non-identical instances of a class normally compare as non-equal unless the class defines the __eq__() method or the __cmp__() method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines either enough of the rich comparison methods (__lt__(), __le__(), __gt__(), and __ge__()) or the __cmp__() method.

CPython implementation detail: Objects of different types except numbers are ordered by their type names; objects of the same types that don't support proper comparison are ordered by their address.

Two more operations with the same syntactic priority, in and not in, are supported only by sequence types (below).

5.4 Numeric Types — int, float, long, complex

There are four distinct numeric types: plain integers, long integers, floating point numbers, and complex numbers. In addition, Booleans are a subtype of plain integers. Plain integers (also just called integers) are implemented using long in C, which gives them at least 32 bits of precision (sys.maxint is always set to the maximum plain integer value for the current platform, the minimum value is -sys.maxint - 1). Long integers have unlimited precision. Floating point numbers are usually implemented using double in C; information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in sys.float_info. Complex numbers have a real and imaginary part, which are each a floating point number. To extract these parts from a complex number z, use z.real and z.imag. (The standard library includes additional numeric types, fractions that hold rationals, and decimal that hold floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including binary, hex, and octal numbers) yield plain integers unless the value they denote is too large to be represented as a plain integer, in which case they yield a long integer. Integer literals with an `'L'` or `'l'` suffix yield long integers (`'L'` is preferred because 1l looks too much like eleven!). Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending `'j'` or `'J'` to a numeric literal yields an imaginary number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the "narrower" type is widened to that of the other, where plain integer is narrower than long integer is narrower than floating point is narrower than complex. Comparisons between numbers of mixed type use the same rule.[2] The constructors int(), long(), float(), and complex() can be used to produce numbers of a specific type.

All built-in numeric types support the following operations. See power and later sections for the operators' priorities.

[2] As a consequence, the list [1, 2] is considered equal to [1.0, 2.0], and similarly for tuples.

Operation	Result	Notes
x + y	sum of x and y	
x - y	difference of x and y	
x * y	product of x and y	
x / y	quotient of x and y	(1)
x // y	(floored) quotient of x and y	(4)(5)
x % y	remainder of x / y	(4)
-x	x negated	
+x	x unchanged	
abs(x)	absolute value or magnitude of x	(3)
int(x)	x converted to integer	(2)
long(x)	x converted to long integer	(2)
float(x)	x converted to floating point	(6)
complex(re,im)	a complex number with real part re, imaginary part im. im defaults to zero.	
c.conjugate()	conjugate of the complex number c. (Identity on real numbers)	
divmod(x, y)	the pair (x // y, x % y)	(3)(4)
pow(x, y)	x to the power y	(3)(7)
x ** y	x to the power y	(7)

Notes:

(1) For (plain or long) integer division, the result is an integer. The result is always rounded towards minus infinity: 1/2 is 0, (-1)/2 is -1, 1/(-2) is -1, and (-1)/(-2) is 0. Note that the result is a long integer if either operand is a long integer, regardless of the numeric value.

(2) Conversion from floats using int() or long() truncates toward zero like the related function, math.trunc(). Use the function math.floor() to round downward and math.ceil() to round upward.

(3) See Built-in Functions for a full description.

(4) Deprecated since version 2.3: The floor division operator, the modulo operator, and the divmod() function are no longer defined for complex numbers. Instead, convert to a floating point number using the abs() function if appropriate.

(5) Also referred to as integer division. The resultant value is a whole integer, though the result's type is not necessarily int.

(6) float also accepts the strings "nan" and "inf" with an optional prefix "+" or "-" for Not a Number (NaN) and positive or negative infinity.

New in version 2.6.

(7) Python defines pow(0, 0) and 0 ** 0 to be 1, as is common for programming languages.

All numbers.Real types (int, long, and float) also include the following operations:

Operation	Result
math.trunc(x)	x truncated to Integral
round(x[, n])	x rounded to n digits, rounding ties away from zero. If n is omitted, it defaults to 0.
math.floor(x)	the greatest integer as a float <= x
math.ceil(x)	the least integer as a float >= x

5.4.1 Bitwise Operations on Integer Types

Bitwise operations only make sense for integers. Negative numbers are treated as their 2's complement value (this assumes a sufficiently large number of bits that no overflow occurs during the operation).

The priorities of the binary bitwise operations are all lower than the numeric operations and higher than the comparisons; the unary operation ~ has the same priority as the other unary numeric operations (+ and -).

This table lists the bitwise operations sorted in ascending priority:

Operation	Result	Notes
x | y	bitwise or of x and y	
x ^ y	bitwise exclusive or of x and y	
x & y	bitwise and of x and y	
x << n	x shifted left by n bits	(1)(2)
x >> n	x shifted right by n bits	(1)(3)
~x	the bits of x inverted	

Notes:

(1) Negative shift counts are illegal and cause a ValueError to be raised.

(2) A left shift by n bits is equivalent to multiplication by pow(2, n). A long integer is returned if the result exceeds the range of plain integers.

(3) A right shift by n bits is equivalent to division by pow(2, n).

5.4.2 Additional Methods on Integer Types

The integer types implement the numbers.Integral abstract base class. In addition, they provide one more method:

int.bit_length()

long.bit_length()

Return the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

More precisely, if x is nonzero, then x.bit_length() is the unique positive integer k such that 2**(k-1) <= abs(x) < 2**k. Equivalently, when abs(x) is small enough to have a correctly rounded logarithm, then k = 1 + int(log(abs(x), 2)). If x is zero, then x.bit_length() returns 0.

Equivalent to:

```
def bit_length(self):
    s = bin(self)       # binary representation:  bin(-37) --> '-0b100101'
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)       # len('100101') --> 6
```

New in version 2.7.

5.4.3 Additional Methods on Float

The float type implements the numbers.Real abstract base class. float also has the following additional methods.

float.as_integer_ratio()
: Return a pair of integers whose ratio is exactly equal to the original float and with a positive denominator. Raises OverflowError on infinities and a ValueError on NaNs.

  New in version 2.6.

float.is_integer()
: Return True if the float instance is finite with integral value, and False otherwise:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

  New in version 2.6.

Two methods support conversion to and from hexadecimal strings. Since Python's floats are stored internally as binary numbers, converting a float to or from a decimal string usually involves a small rounding error. In contrast, hexadecimal strings allow exact representation and specification of floating-point numbers. This can be useful when debugging, and in numerical work.

float.hex()
: Return a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this representation will always include a leading 0x and a trailing p and exponent.

  New in version 2.6.

float.fromhex(s)
: Class method to return the float represented by a hexadecimal string s. The string s may have leading and trailing whitespace.

  New in version 2.6.

Note that float.hex() is an instance method, while float.fromhex() is a class method.

A hexadecimal string takes the form:

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

where the optional sign may by either + or -, integer and fraction are strings of hexadecimal digits, and exponent is a decimal integer with an optional leading sign. Case is not significant, and there must be at least one hexadecimal digit in either the integer or the fraction. This syntax is similar to the syntax specified in section 6.4.4.2 of the C99 standard, and also to the syntax used in Java 1.5 onwards. In particular, the output of float.hex() is usable as a hexadecimal floating-point literal in C or Java code, and hexadecimal strings produced by C's %a format character or Java's Double.toHexString are accepted by float.fromhex().

Note that the exponent is written in decimal rather than hexadecimal, and that it gives the power of 2 by which to multiply the coefficient. For example, the hexadecimal string 0x3.a7p10 represents the floating-point number (3 + 10./16 + 7./16**2) * 2.0**10, or 3740.0:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

Applying the reverse conversion to 3740.0 gives a different hexadecimal string representing the same number:

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

5.5 Iterator Types

New in version 2.2.

Python supports a concept of iteration over containers. This is implemented using two distinct methods; these are used to allow user-defined classes to support iteration. Sequences, described below in more detail, always support the iteration methods.

One method needs to be defined for container objects to provide iteration support:

container.__iter__()
: Return an iterator object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types. (An example of an object supporting multiple forms of iteration would be a tree structure which supports both breadth-first and depth-first traversal.) This method corresponds to the tp_iter slot of the type structure for Python objects in the Python/C API.

The iterator objects themselves are required to support the following two methods, which together form the iterator protocol:

iterator.__iter__()
: Return the iterator object itself. This is required to allow both containers and iterators to be used with the for and in statements. This method corresponds to the tp_iter slot of the type structure for Python objects in the Python/C API.

iterator.next()
: Return the next item from the container. If there are no further items, raise the StopIteration exception. This method corresponds to the tp_iternext slot of the type structure for Python objects in the Python/C API.

Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

The intention of the protocol is that once an iterator's next() method raises StopIteration, it will continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken. (This constraint was added in Python 2.3; in Python 2.2, various iterators are broken according to this rule.)

5.5.1 Generator Types

Python's generators provide a convenient way to implement the iterator protocol. If a container object's __iter__() method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the __iter__() and next() methods. More information about generators can be found in the documentation for the yield expression.

5.6 Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange

There are seven sequence types: strings, Unicode strings, lists, tuples, bytearrays, buffers, and xrange objects.

For other containers see the built in dict and set classes, and the collections module.

String literals are written in single or double quotes: 'xyzzy', "frobozz". See strings for more about string literals. Unicode strings are much like strings, but are specified in the syntax using a preceding 'u' character: u'abc', u"def". In addition to the functionality described here, there are also string-specific methods described in the String Methods section. Lists are constructed with square brackets, separating items with commas: [a, b, c]. Tuples are constructed by the comma operator (not within square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, such as a, b, c or (). A single item tuple must have a trailing comma, such as (d,).

Bytearray objects are created with the built-in function bytearray().

Buffer objects are not directly supported by Python syntax, but can be created by calling the built-in function buffer(). They don't support concatenation or repetition.

Objects of type xrange are similar to buffers in that there is no specific syntax to create them, but they are created using the xrange() function. They don't support slicing, concatenation or repetition, and using in, not in, min() or max() on them is inefficient.

Most sequence types support the following operations. The in and not in operations have the same priorities as the comparison operations. The + and * operations have the same priority as the corresponding numeric operations.[3] Additional methods are provided for Mutable Sequence Types.

This table lists the sequence operations sorted in ascending priority. In the table, s and t are sequences of the same type; n, i and j are integers:

Operation	Result	Notes
x in s	True if an item of s is equal to x, else False	(1)
x not in s	False if an item of s is equal to x, else True	(1)
s + t	the concatenation of s and t	(6)
s * n, n * s	equivalent to adding s to itself n times	(2)
s[i]	ith item of s, origin 0	(3)
s[i:j]	slice of s from i to j	(3)(4)
s[i:j:k]	slice of s from i to j with step k	(3)(5)
len(s)	length of s	
min(s)	smallest item of s	
max(s)	largest item of s	
s.index(x)	index of the first occurrence of x in s	
s.count(x)	total number of occurrences of x in s	

Sequence types also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see comparisons in the language reference.)

Notes:

(1) When s is a string or Unicode string object the in and not in operations act like a substring test. In Python versions before 2.3, x had to be a string of length 1. In Python 2.3 and beyond, x may be a string of any length.

(2) Values of n less than 0 are treated as 0 (which yields an empty sequence of the same type as s). Note that items in the sequence s are not copied; they are referenced multiple times. This often haunts new Python programmers; consider:

```
>>> lists = [[]] * 3
>>> lists
```

(continues on next page)

[3] They must have since the parser can't tell the type of the operands.

(continued from previous page)

```
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

What has happened is that [[]] is a one-element list containing an empty list, so all three elements of [[]] * 3 are references to this single empty list. Modifying any of the elements of lists modifies this single list. You can create a list of different lists this way:

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

Further explanation is available in the FAQ entry faq-multidimensional-list.

(3) If i or j is negative, the index is relative to the end of sequence s: len(s) + i or len(s) + j is substituted. But note that -0 is still 0.

(4) The slice of s from i to j is defined as the sequence of items with index k such that i <= k < j. If i or j is greater than len(s), use len(s). If i is omitted or None, use 0. If j is omitted or None, use len(s). If i is greater than or equal to j, the slice is empty.

(5) The slice of s from i to j with step k is defined as the sequence of items with index x = i + n*k such that 0 <= n < (j-i)/k. In other words, the indices are i, i+k, i+2*k, i+3*k and so on, stopping when j is reached (but never including j). When k is positive, i and j are reduced to len(s) if they are greater. When k is negative, i and j are reduced to len(s) - 1 if they are greater. If i or j are omitted or None, they become "end" values (which end depends on the sign of k). Note, k cannot be zero. If k is None, it is treated like 1.

(6) CPython implementation detail: If s and t are both strings, some Python implementations such as CPython can usually perform an in-place optimization for assignments of the form s = s + t or s += t. When applicable, this optimization makes quadratic run-time much less likely. This optimization is both version and implementation dependent. For performance sensitive code, it is preferable to use the str.join() method which assures consistent linear concatenation performance across versions and implementations.

Changed in version 2.4: Formerly, string concatenation never occurred in-place.

5.6.1 String Methods

Below are listed the string methods which both 8-bit strings and Unicode objects support. Some of them are also available on bytearray objects.

In addition, Python's strings support the sequence type methods described in the Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange section. To output formatted strings use template strings or the % operator described in the String Formatting Operations section. Also, see the re module for string functions based on regular expressions.

str.capitalize()

Return a copy of the string with its first character capitalized and the rest lowercased.

For 8-bit strings, this method is locale-dependent.

str.center(width[, fillchar])
: Return centered in a string of length width. Padding is done using the specified fillchar (default is a space).

  Changed in version 2.4: Support for the fillchar argument.

str.count(sub[, start[, end]])
: Return the number of non-overlapping occurrences of substring sub in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.

str.decode([encoding[, errors]])
: Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding. errors may be given to set a different error handling scheme. The default is `'strict'`, meaning that encoding errors raise UnicodeError. Other possible values are `'ignore'`, `'replace'` and any other name registered via codecs.register_error(), see section Codec Base Classes.

  New in version 2.2.

  Changed in version 2.3: Support for other error handling schemes added.

  Changed in version 2.7: Support for keyword arguments added.

str.encode([encoding[, errors]])
: Return an encoded version of the string. Default encoding is the current default string encoding. errors may be given to set a different error handling scheme. The default for errors is `'strict'`, meaning that encoding errors raise a UnicodeError. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` and any other name registered via codecs.register_error(), see section Codec Base Classes. For a list of possible encodings, see section Standard Encodings.

  New in version 2.0.

  Changed in version 2.3: Support for `'xmlcharrefreplace'` and `'backslashreplace'` and other error handling schemes added.

  Changed in version 2.7: Support for keyword arguments added.

str.endswith(suffix[, start[, end]])
: Return True if the string ends with the specified suffix, otherwise return False. suffix can also be a tuple of suffixes to look for. With optional start, test beginning at that position. With optional end, stop comparing at that position.

  Changed in version 2.5: Accept tuples as suffix.

str.expandtabs([tabsize])
: Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every tabsize characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (\t), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (\n) or return (\r), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed.

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

str.find(sub[, start[, end]])
: Return the lowest index in the string where substring sub is found within the slice s[start:end]. Optional

arguments start and end are interpreted as in slice notation. Return -1 if sub is not found.

Note: The find() method should be used only if you need to know the position of sub. To check if sub is a substring or not, use the in operator:

```
>>> 'Py' in 'Python'
True
```

---

str.format(*args, **kwargs)

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces {}. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

See Format String Syntax for a description of the various formatting options that can be specified in format strings.

This method of string formatting is the new standard in Python 3, and should be preferred to the % formatting described in String Formatting Operations in new code.

New in version 2.6.

str.index(sub[, start[, end]])

Like find(), but raise ValueError when the substring is not found.

str.isalnum()

Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str.isalpha()

Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str.isdigit()

Return true if all characters in the string are digits and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str.islower()

Return true if all cased characters[4] in the string are lowercase and there is at least one cased character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str.isspace()

Return true if there are only whitespace characters in the string and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

[4] Cased characters are those with general category property being one of "Lu" (Letter, uppercase), "Ll" (Letter, lowercase), or "Lt" (Letter, titlecase).

str.istitle()
: Return true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

  For 8-bit strings, this method is locale-dependent.

str.isupper()
: Return true if all cased characters[4] in the string are uppercase and there is at least one cased character, false otherwise.

  For 8-bit strings, this method is locale-dependent.

str.join(iterable)
: Return a string which is the concatenation of the strings in iterable. If there is any Unicode object in iterable, return a Unicode instead. A TypeError will be raised if there are any non-string or non Unicode object values in iterable. The separator between elements is the string providing this method.

str.ljust(width[, fillchar])
: Return the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than or equal to len(s).

  Changed in version 2.4: Support for the fillchar argument.

str.lower()
: Return a copy of the string with all the cased characters[4] converted to lowercase.

  For 8-bit strings, this method is locale-dependent.

str.lstrip([chars])
: Return a copy of the string with leading characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

  Changed in version 2.2.2: Support for the chars argument.

str.partition(sep)
: Split the string at the first occurrence of sep, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

  New in version 2.5.

str.replace(old, new[, count])
: Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

str.rfind(sub[, start[, end]])
: Return the highest index in the string where substring sub is found, such that sub is contained within s[start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

str.rindex(sub[, start[, end]])
: Like rfind() but raises ValueError when the substring sub is not found.

str.rjust(width[, fillchar])
: Return the string right justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than or equal to len(s).

  Changed in version 2.4: Support for the fillchar argument.

str.rpartition(sep)
: Split the string at the last occurrence of sep, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

  New in version 2.5.

str.rsplit([sep[, maxsplit]])
: Return a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done, the rightmost ones. If sep is not specified or None, any whitespace string is a separator. Except for splitting from the right, rsplit() behaves like split() which is described in detail below.

  New in version 2.4.

str.rstrip([chars])
: Return a copy of the string with trailing characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

  Changed in version 2.2.2: Support for the chars argument.

str.split([sep[, maxsplit]])
: Return a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done (thus, the list will have at most maxsplit+1 elements). If maxsplit is not specified or -1, then there is no limit on the number of splits (all possible splits are made).

  If sep is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, '1,,2'.split(',') returns ['1', '', '2']). The sep argument may consist of multiple characters (for example, '1<>2<>3'.split('<>') returns ['1', '2', '3']). Splitting an empty string with a specified separator returns [''].

  If sep is not specified or is None, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a None separator returns [].

  For example, ' 1 2 3 '.split() returns ['1', '2', '3'], and ' 1 2 3 '.split(None, 1) returns ['1', '2 3 '].

str.splitlines([keepends])
: Return a list of the lines in the string, breaking at line boundaries. This method uses the universal newlines approach to splitting lines. Line breaks are not included in the resulting list unless keepends is given and true.

  Python recognizes "\r", "\n", and "\r\n" as line boundaries for 8-bit strings.

  For example:

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

Unlike split() when a delimiter string sep is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

For comparison, split('\n') gives:

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

unicode.splitlines([keepends])

Return a list of the lines in the string, like str.splitlines(). However, the Unicode method splits on the following line boundaries, which are a superset of the universal newlines recognized for 8-bit strings.

Representation	Description
\n	Line Feed
\r	Carriage Return
\r\n	Carriage Return + Line Feed
\v or \x0b	Line Tabulation
\f or \x0c	Form Feed
\x1c	File Separator
\x1d	Group Separator
\x1e	Record Separator
\x85	Next Line (C1 Control Code)
\u2028	Line Separator
\u2029	Paragraph Separator

Changed in version 2.7: \v and \f added to list of line boundaries.

str.startswith(prefix[, start[, end]])

Return True if string starts with the prefix, otherwise return False. prefix can also be a tuple of prefixes to look for. With optional start, test string beginning at that position. With optional end, stop comparing string at that position.

Changed in version 2.5: Accept tuples as prefix.

str.strip([chars])

Return a copy of the string with the leading and trailing characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.strip()
'spacious'
```

(continues on next page)

(continued from previous page)

```
>>> 'www.example.com'.strip('cmowz.')
'example'
```

Changed in version 2.2.2: Support for the chars argument.

str.swapcase()
Return a copy of the string with uppercase characters converted to lowercase and vice versa.

For 8-bit strings, this method is locale-dependent.

str.title()
Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```
>>> "they're bill's friends from the UK".title()
"They're Bill's Friends From The Uk"
```

A workaround for apostrophes can be constructed using regular expressions:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

For 8-bit strings, this method is locale-dependent.

str.translate(table[, deletechars])
Return a copy of the string where all characters occurring in the optional argument deletechars are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256.

You can use the maketrans() helper function in the string module to create a translation table. For string objects, set the table argument to None for translations that only delete characters:

```
>>> 'read this short text'.translate(None, 'aeiou')
'rd ths shrt txt'
```

New in version 2.6: Support for a None table argument.

For Unicode objects, the translate() method does not accept the optional deletechars argument. Instead, it returns a copy of the s where all characters have been mapped through the given translation table which must be a mapping of Unicode ordinals to Unicode ordinals, Unicode strings or None. Unmapped characters are left untouched. Characters mapped to None are deleted. Note, a more flexible approach is to create a custom character mapping codec using the codecs module (see encodings.cp1251 for an example).

str.upper()
Return a copy of the string with all the cased characters[4] converted to uppercase. Note that s.upper().

isupper() might be False if s contains uncased characters or if the Unicode category of the resulting character(s) is not "Lu" (Letter, uppercase), but e.g. "Lt" (Letter, titlecase).

For 8-bit strings, this method is locale-dependent.

str.zfill(width)
Return the numeric string left filled with zeros in a string of length width. A sign prefix is handled correctly. The original string is returned if width is less than or equal to len(s).

New in version 2.2.2.

The following methods are present only on unicode objects:

unicode.isnumeric()
Return True if there are only numeric characters in S, False otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH.

unicode.isdecimal()
Return True if there are only decimal characters in S, False otherwise. Decimal characters include digit characters, and all characters that can be used to form decimal-radix numbers, e.g. U+0660, ARABIC-INDIC DIGIT ZERO.

5.6.2 String Formatting Operations

String and Unicode objects have one unique built-in operation: the % operator (modulo). This is also known as the string formatting or interpolation operator. Given format % values (where format is a string or Unicode object), % conversion specifications in format are replaced with zero or more elements of values. The effect is similar to the using sprintf() in the C language. If format is a Unicode object, or if any of the objects being converted using the %s conversion are Unicode objects, the result will also be a Unicode object.

If format requires a single argument, values may be a single non-tuple object.[5] Otherwise, values must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The '%' character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, (somename)).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an '*' (asterisk), the actual width is read from the next element of the tuple in values, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a '.' (dot) followed by the precision. If specified as '*' (an asterisk), the actual width is read from the next element of the tuple in values, and the value to convert comes after the precision.
6. Length modifier (optional).
7. Conversion type.

[5] To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.

When the right argument is a dictionary (or other mapping type), then the formats in the string must include a parenthesised mapping key into that dictionary inserted immediately after the '%' character. The mapping key selects the value to be formatted from the mapping. For example:

```
>>> print '%(language)s has %(number)03d quote types.' % \
...       {"language": "Python", "number": 2}
Python has 002 quote types.
```

In this case no * specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	Meaning
'#'	The value conversion will use the "alternate form" (where defined below).
'0'	The conversion will be zero padded for numeric values.
'-'	The converted value is left adjusted (overrides the '0' conversion if both are given).
' '	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
'+'	A sign character ('+' or '-') will precede the conversion (overrides a "space" flag).

A length modifier (h, l, or L) may be present, but is ignored as it is not necessary for Python – so e.g. %ld is identical to %d.

The conversion types are:

Conversion	Meaning	Notes
'd'	Signed integer decimal.	
'i'	Signed integer decimal.	
'o'	Signed octal value.	(1)
'u'	Obsolete type – it is identical to 'd'.	(7)
'x'	Signed hexadecimal (lowercase).	(2)
'X'	Signed hexadecimal (uppercase).	(2)
'e'	Floating point exponential format (lowercase).	(3)
'E'	Floating point exponential format (uppercase).	(3)
'f'	Floating point decimal format.	(3)
'F'	Floating point decimal format.	(3)
'g'	Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'G'	Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'c'	Single character (accepts integer or single character string).	
'r'	String (converts any Python object using repr()).	(5)
's'	String (converts any Python object using str()).	(6)
'%'	No argument is converted, results in a '%' character in the result.	

Notes:

(1) The alternate form causes a leading zero ('0') to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.

(2) The alternate form causes a leading '0x' or '0X' (depending on whether the 'x' or 'X' format was used) to be inserted before the first digit.

(3) The alternate form causes the result to always contain a decimal point, even if no digits follow it.

The precision determines the number of digits after the decimal point and defaults to 6.

(4) The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

The precision determines the number of significant digits before and after the decimal point and defaults to 6.

(5) The %r conversion was added in Python 2.0.

The precision determines the maximal number of characters used.

(6) If the object or format provided is a unicode string, the resulting string will also be unicode.

The precision determines the maximal number of characters used.

(7) See PEP 237.

Since Python strings have an explicit length, %s conversions do not assume that `'\0'` is the end of the string.

Changed in version 2.7: %f conversions for numbers whose absolute value is over 1e50 are no longer replaced by %g conversions.

Additional string operations are defined in standard modules string and re.

5.6.3 XRange Type

The xrange type is an immutable sequence which is commonly used for looping. The advantage of the xrange type is that an xrange object will always take the same amount of memory, no matter the size of the range it represents. There are no consistent performance advantages.

XRange objects have very little behavior: they only support indexing, iteration, and the len() function.

5.6.4 Mutable Sequence Types

List and bytearray objects support additional operations that allow in-place modification of the object. Other mutable sequence types (when added to the language) should also support these operations. Strings and tuples are immutable sequence types: such objects cannot be modified once created. The following operations are defined on mutable sequence types (where x is an arbitrary object):

Operation	Result	Notes
s[i] = x	item i of s is replaced by x	
s[i:j] = t	slice of s from i to j is replaced by the contents of the iterable t	
del s[i:j]	same as s[i:j] = []	
s[i:j:k] = t	the elements of s[i:j:k] are replaced by those of t	(1)
del s[i:j:k]	removes the elements of s[i:j:k] from the list	
s.append(x)	same as s[len(s):len(s)] = [x]	(2)
s.extend(t) or s += t	for the most part the same as s[len(s):len(s)] = t	(3)
s *= n	updates s with its contents repeated n times	(11)
s.count(x)	return number of i's for which s[i] == x	
s.index(x[, i[, j]])	return smallest k such that s[k] == x and i <= k < j	(4)
s.insert(i, x)	same as s[i:i] = [x]	(5)
s.pop([i])	same as x = s[i]; del s[i]; return x	(6)
s.remove(x)	same as del s[s.index(x)]	(4)
s.reverse()	reverses the items of s in place	(7)
s.sort([cmp[, key[, reverse]]])	sort the items of s in place	(7)(8)(9)(10)

Notes:

(1) t must have the same length as the slice it is replacing.

(2) The C implementation of Python has historically accepted multiple parameters and implicitly joined them into a tuple; this no longer works in Python 2.0. Use of this misfeature has been deprecated since Python 1.4.

(3) t can be any iterable object.

(4) Raises ValueError when x is not found in s. When a negative index is passed as the second or third parameter to the index() method, the list length is added, as for slice indices. If it is still negative, it is truncated to zero, as for slice indices.

Changed in version 2.3: Previously, index() didn't have arguments for specifying start and stop positions.

(5) When a negative index is passed as the first parameter to the insert() method, the list length is added, as for slice indices. If it is still negative, it is truncated to zero, as for slice indices.

Changed in version 2.3: Previously, all negative indices were truncated to zero.

(6) The pop() method's optional argument i defaults to -1, so that by default the last item is removed and returned.

(7) The sort() and reverse() methods modify the list in place for economy of space when sorting or reversing a large list. To remind you that they operate by side effect, they don't return the sorted or reversed list.

(8) The sort() method takes optional arguments for controlling the comparisons.

cmp specifies a custom comparison function of two arguments (list items) which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument: cmp=lambda x,y: cmp(x.lower(), y.lower()). The default value is None.

key specifies a function of one argument that is used to extract a comparison key from each list element: key=str.lower. The default value is None.

reverse is a boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

In general, the key and reverse conversion processes are much faster than specifying an equivalent cmp function. This is because cmp is called multiple times for each list element while key and reverse touch each element only once. Use functools.cmp_to_key() to convert an old-style cmp function to a key function.

Changed in version 2.3: Support for None as an equivalent to omitting cmp was added.

Changed in version 2.4: Support for key and reverse was added.

(9) Starting with Python 2.3, the sort() method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

(10) CPython implementation detail: While a list is being sorted, the effect of attempting to mutate, or even inspect, the list is undefined. The C implementation of Python 2.3 and newer makes the list appear empty for the duration, and raises ValueError if it can detect that the list has been mutated during a sort.

(11) The value n is an integer, or an object implementing __index__(). Zero and negative values of n clear the sequence. Items in the sequence are not copied; they are referenced multiple times, as explained for s * n under Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange.

5.7 Set Types — set, frozenset

A set object is an unordered collection of distinct hashable objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. (For other containers see the built in dict, list, and tuple classes, and the collections module.)

New in version 2.4.

Like other collections, sets support x in set, len(set), and for x in set. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

There are currently two built-in set types, set and frozenset. The set type is mutable — the contents can be changed using methods like add() and remove(). Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The frozenset type is immutable and hashable — its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

As of Python 2.7, non-empty sets (not frozensets) can be created by placing a comma-separated list of elements within braces, for example: {'jack', 'sjoerd'}, in addition to the set constructor.

The constructors for both classes work the same:

class set([iterable])

class frozenset([iterable])

Return a new set or frozenset object whose elements are taken from iterable. The elements of a set must be hashable. To represent sets of sets, the inner sets must be frozenset objects. If iterable is not specified, a new empty set is returned.

Instances of set and frozenset provide the following operations:

len(s)

Return the number of elements in set s (cardinality of s).

x in s
: Test x for membership in s.

x not in s
: Test x for non-membership in s.

isdisjoint(other)
: Return True if the set has no elements in common with other. Sets are disjoint if and only if their intersection is the empty set.

  New in version 2.6.

issubset(other)
set <= other
: Test whether every element in the set is in other.

set < other
: Test whether the set is a proper subset of other, that is, set <= other and set != other.

issuperset(other)
set >= other
: Test whether every element in other is in the set.

set > other
: Test whether the set is a proper superset of other, that is, set >= other and set != other.

union(*others)
set | other | ...
: Return a new set with elements from the set and all others.

  Changed in version 2.6: Accepts multiple input iterables.

intersection(*others)
set & other & ...
: Return a new set with elements common to the set and all others.

  Changed in version 2.6: Accepts multiple input iterables.

difference(*others)
set - other - ...
: Return a new set with elements in the set that are not in the others.

  Changed in version 2.6: Accepts multiple input iterables.

symmetric_difference(other)
set ^ other
: Return a new set with elements in either the set or other but not both.

copy()
: Return a new set with a shallow copy of s.

Note, the non-operator versions of union(), intersection(), difference(), and symmetric_difference(), issubset(), and issuperset() methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like set('abc') & 'cbs' in favor of the more readable set('abc').intersection('cbs').

Both set and frozenset support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Instances of set are compared to instances of frozenset based on their members. For example, set('abc') == frozenset('abc') returns True and so does set('abc') in set([frozenset('abc')]).

The subset and equality comparisons do not generalize to a total ordering function. For example, any two non-empty disjoint sets are not equal and are not subsets of each other, so all of the following return False: a<b, a==b, or a>b. Accordingly, sets do not implement the __cmp__() method.

Since sets only define partial ordering (subset relationships), the output of the list.sort() method is undefined for lists of sets.

Set elements, like dictionary keys, must be hashable.

Binary operations that mix set instances with frozenset return the type of the first operand. For example: frozenset('ab') | set('bc') returns an instance of frozenset.

The following table lists operations available for set that do not apply to immutable instances of frozenset:

update(*others)
set |= other | ...
: Update the set, adding elements from all others.

  Changed in version 2.6: Accepts multiple input iterables.

intersection_update(*others)
set &= other & ...
: Update the set, keeping only elements found in it and all others.

  Changed in version 2.6: Accepts multiple input iterables.

difference_update(*others)
set -= other | ...
: Update the set, removing elements found in others.

  Changed in version 2.6: Accepts multiple input iterables.

symmetric_difference_update(other)
set ^= other
: Update the set, keeping only elements found in either set, but not in both.

add(elem)
: Add element elem to the set.

remove(elem)
: Remove element elem from the set. Raises KeyError if elem is not contained in the set.

discard(elem)
: Remove element elem from the set if it is present.

pop()
: Remove and return an arbitrary element from the set. Raises KeyError if the set is empty.

clear()
: Remove all elements from the set.

Note, the non-operator versions of the update(), intersection_update(), difference_update(), and symmetric_difference_update() methods will accept any iterable as an argument.

Note, the elem argument to the __contains__(), remove(), and discard() methods may be a set. To support searching for an equivalent frozenset, a temporary one is created from elem.

See also:

Comparison to the built-in set types Differences between the sets module and the built-in set types.

5.8 Mapping Types — dict

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary. (For other containers see the built in list, set, and tuple classes, and the collections module.)

A dictionary's keys are almost arbitrary values. Values that are not hashable, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as 1 and 1.0) then they can be used interchangeably to index the same dictionary entry. (Note however, that since computers store floating-point numbers as approximations it is usually unwise to use them as dictionary keys.)

Dictionaries can be created by placing a comma-separated list of key: value pairs within braces, for example: {'jack': 4098, 'sjoerd': 4127} or {4098: 'jack', 4127: 'sjoerd'}, or by the dict constructor.

class dict(**kwarg)
class dict(mapping, **kwarg)
class dict(iterable, **kwarg)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object. Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable with exactly two objects. The first object of each item becomes a key in the new dictionary, and the second object the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

To illustrate, the following examples all return a dictionary equal to {"one": 1, "two": 2, "three": 3}:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

New in version 2.2.

Changed in version 2.3: Support for building a dictionary from keyword arguments added.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

len(d)
Return the number of items in the dictionary d.

d[key]
Return the item of d with key key. Raises a KeyError if key is not in the map.

If a subclass of dict defines a method __missing__() and key is not present, the d[key] operation calls that method with the key key as argument. The d[key] operation then returns or raises whatever is returned or raised by the __missing__(key) call. No other operations or methods invoke __missing__(). If __missing__() is not defined, KeyError is raised. __missing__() must be a method; it cannot be an instance variable:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

The example above shows part of the implementation of collections.Counter. A different __missing__ method is used by collections.defaultdict.

New in version 2.5: Recognition of __missing__ methods of dict subclasses.

d[key] = value
Set d[key] to value.

del d[key]
Remove d[key] from d. Raises a KeyError if key is not in the map.

key in d
Return True if d has a key key, else False.

New in version 2.2.

key not in d
Equivalent to not key in d.

New in version 2.2.

iter(d)
Return an iterator over the keys of the dictionary. This is a shortcut for iterkeys().

clear()
Remove all items from the dictionary.

copy()
Return a shallow copy of the dictionary.

fromkeys(seq[, value])
Create a new dictionary with keys from seq and values set to value.

fromkeys() is a class method that returns a new dictionary. value defaults to None.

New in version 2.3.

get(key[, default])
Return the value for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.

has_key(key)
Test for the presence of key in the dictionary. has_key() is deprecated in favor of key in d.

items()
Return a copy of the dictionary's list of (key, value) pairs.

CPython implementation detail: Keys and values are listed in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions.

If items(), keys(), values(), iteritems(), iterkeys(), and itervalues() are called with no intervening modifications to the dictionary, the lists will directly correspond. This allows the creation of (value, key) pairs using zip(): pairs = zip(d.values(), d.keys()). The same relationship holds for the iterkeys() and itervalues() methods: pairs = zip(d.itervalues(), d.iterkeys()) provides the same value for pairs. Another way to create the same list is pairs = [(v, k) for (k, v) in d.iteritems()].

iteritems()
Return an iterator over the dictionary's (key, value) pairs. See the note for dict.items().

Using iteritems() while adding or deleting entries in the dictionary may raise a RuntimeError or fail to iterate over all entries.

New in version 2.2.

iterkeys()
Return an iterator over the dictionary's keys. See the note for dict.items().

Using iterkeys() while adding or deleting entries in the dictionary may raise a RuntimeError or fail to iterate over all entries.

New in version 2.2.

itervalues()
Return an iterator over the dictionary's values. See the note for dict.items().

Using itervalues() while adding or deleting entries in the dictionary may raise a RuntimeError or fail to iterate over all entries.

New in version 2.2.

keys()
Return a copy of the dictionary's list of keys. See the note for dict.items().

pop(key[, default])
If key is in the dictionary, remove it and return its value, else return default. If default is not given and key is not in the dictionary, a KeyError is raised.

New in version 2.3.

popitem()
Remove and return an arbitrary (key, value) pair from the dictionary.

popitem() is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling popitem() raises a KeyError.

setdefault(key[, default])
If key is in the dictionary, return its value. If not, insert key with a value of default and return default. default defaults to None.

update([other])
Update the dictionary with the key/value pairs from other, overwriting existing keys. Return None.

update() accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: d.update(red=1, blue=2).

Changed in version 2.4: Allowed the argument to be an iterable of key/value pairs and allowed keyword arguments.

values()
Return a copy of the dictionary's list of values. See the note for dict.items().

viewitems()
Return a new view of the dictionary's items ((key, value) pairs). See below for documentation of view objects.

New in version 2.7.

viewkeys()
Return a new view of the dictionary's keys. See below for documentation of view objects.

New in version 2.7.

viewvalues()
Return a new view of the dictionary's values. See below for documentation of view objects.

New in version 2.7.

Dictionaries compare equal if and only if they have the same (key, value) pairs.

5.8.1 Dictionary view objects

The objects returned by dict.viewkeys(), dict.viewvalues() and dict.viewitems() are view objects. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests:

len(dictview)
Return the number of entries in the dictionary.

iter(dictview)
Return an iterator over the keys, values or items (represented as tuples of (key, value)) in the dictionary.

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond. This allows the creation of (value, key) pairs using zip(): pairs = zip(d.values(), d.keys()). Another way to create the same list is pairs = [(v, k) for (k, v) in d.items()].

Iterating views while adding or deleting entries in the dictionary may raise a RuntimeError or fail to iterate over all entries.

x in dictview
Return True if x is in the underlying dictionary's keys, values or items (in the latter case, x should be a (key, value) tuple).

Keys views are set-like since their entries are unique and hashable. If all values are hashable, so that (key, value) pairs are unique and hashable, then the items view is also set-like. (Values views are not treated as set-like since the entries are generally not unique.) Then these set operations are available ("other" refers either to another view or a set):

dictview & other
Return the intersection of the dictview and the other object as a new set.

dictview | other
Return the union of the dictview and the other object as a new set.

dictview - other
Return the difference between the dictview and the other object (all elements in dictview that aren't in other) as a new set.

dictview ^ other
: Return the symmetric difference (all elements either in dictview or other, but not in both) of the dictview and the other object as a new set.

An example of dictionary view usage:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.viewkeys()
>>> values = dishes.viewvalues()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order
>>> list(keys)
['eggs', 'bacon', 'sausage', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['spam', 'bacon']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
```

5.9 File Objects

File objects are implemented using C's stdio package and can be created with the built-in open() function. File objects are also returned by some other built-in functions and methods, such as os.popen() and os.fdopen() and the makefile() method of socket objects. Temporary files can be created using the tempfile module, and high-level file operations such as copying, moving, and deleting files and directories can be achieved with the shutil module.

When a file operation fails for an I/O-related reason, the exception IOError is raised. This includes situations where the operation is not defined for some reason, like seek() on a tty device or writing a file opened for reading.

Files have the following methods:

file.close()
: Close the file. A closed file cannot be read or written any more. Any operation which requires that the file be open will raise a ValueError after the file has been closed. Calling close() more than once is allowed.

  As of Python 2.5, you can avoid having to call this method explicitly if you use the with statement. For example, the following code will automatically close f when the with block is exited:

```
from __future__ import with_statement # This isn't required in Python 2.6

with open("hello.txt") as f:
    for line in f:
        print line,
```

In older versions of Python, you would have needed to do this to get the same effect:

```
f = open("hello.txt")
try:
    for line in f:
        print line,
finally:
    f.close()
```

Note: Not all "file-like" types in Python support use as a context manager for the with statement. If your code is intended to work with any file-like object, you can use the function contextlib.closing() instead of using the object directly.

file.flush()
Flush the internal buffer, like stdio's fflush(). This may be a no-op on some file-like objects.

Note: flush() does not necessarily write the file's data to disk. Use flush() followed by os.fsync() to ensure this behavior.

file.fileno()
Return the integer "file descriptor" that is used by the underlying implementation to request I/O operations from the operating system. This can be useful for other, lower level interfaces that use file descriptors, such as the fcntl module or os.read() and friends.

Note: File-like objects which do not have a real file descriptor should not provide this method!

file.isatty()
Return True if the file is connected to a tty(-like) device, else False.

Note: If a file-like object is not associated with a real file, this method should not be implemented.

file.next()
A file object is its own iterator, for example iter(f) returns f (unless f is closed). When a file is used as an iterator, typically in a for loop (for example, for line in f: print line.strip()), the next() method is called repeatedly. This method returns the next input line, or raises StopIteration when EOF is hit when the file is open for reading (behavior is undefined when the file is open for writing). In order to make a for loop the most efficient way of looping over the lines of a file (a very common operation), the next() method uses a hidden read-ahead buffer. As a consequence of using a read-ahead buffer, combining next() with other file methods (like readline()) does not work right. However, using seek() to reposition the file to an absolute position will flush the read-ahead buffer.

New in version 2.3.

file.read([size])
Read at most size bytes from the file (less if the read hits EOF before obtaining size bytes). If the

size argument is negative or omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately. (For certain files, like ttys, it makes sense to continue reading after an EOF is hit.) Note that this method may call the underlying C function fread() more than once in an effort to acquire as close to size bytes as possible. Also note that when in non-blocking mode, less data than was requested may be returned, even if no size parameter was given.

Note: This function is simply a wrapper for the underlying fread() C function, and will behave the same in corner cases, such as whether the EOF value is cached.

---

file.readline([size])

Read one entire line from the file. A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line).[6] If the size argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. When size is not 0, an empty string is returned only when EOF is encountered immediately.

Note: Unlike stdio's fgets(), the returned string contains null characters (`'\0'`) if they occurred in the input.

---

file.readlines([sizehint])

Read until EOF using readline() and return a list containing the lines thus read. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read. Objects implementing a file-like interface may choose to ignore sizehint if it cannot be implemented, or cannot be implemented efficiently.

file.xreadlines()

This method returns the same thing as iter(f).

New in version 2.1.

Deprecated since version 2.3: Use for line in file instead.

file.seek(offset[, whence])

Set the file's current position, like stdio's fseek(). The whence argument is optional and defaults to os.SEEK_SET or 0 (absolute file positioning); other values are os.SEEK_CUR or 1 (seek relative to the current position) and os.SEEK_END or 2 (seek relative to the file's end). There is no return value.

For example, f.seek(2, os.SEEK_CUR) advances the position by two and f.seek(-3, os.SEEK_END) sets the position to the third to last.

Note that if the file is opened for appending (mode `'a'` or `'a+'`), any seek() operations will be undone at the next write. If the file is only opened for writing in append mode (mode `'a'`), this method is essentially a no-op, but it remains useful for files opened in append mode with reading enabled (mode `'a+'`). If the file is opened in text mode (without `'b'`), only offsets returned by tell() are legal. Use of other offsets causes undefined behavior.

Note that not all file objects are seekable.

Changed in version 2.6: Passing float values as offset has been deprecated.

file.tell()

Return the file's current position, like stdio's ftell().

[6] The advantage of leaving the newline on is that returning an empty string is then an unambiguous EOF indication. It is also possible (in cases where it might matter, for example, if you want to make an exact copy of a file while scanning its lines) to tell whether the last line of a file ended in a newline or not (yes this happens!).

Note: On Windows, tell() can return illegal values (after an fgets()) when reading files with Unix-style line-endings. Use binary mode (`'rb'`) to circumvent this problem.

file.truncate([size])
: Truncate the file's size. If the optional size argument is present, the file is truncated to (at most) that size. The size defaults to the current position. The current file position is not changed. Note that if a specified size exceeds the file's current size, the result is platform-dependent: possibilities include that the file may remain unchanged, increase to the specified size as if zero-filled, or increase to the specified size with undefined new content. Availability: Windows, many Unix variants.

file.write(str)
: Write a string to the file. There is no return value. Due to buffering, the string may not actually show up in the file until the flush() or close() method is called.

file.writelines(sequence)
: Write a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value. (The name is intended to match readlines(); writelines() does not add line separators.)

Files support the iterator protocol. Each iteration returns the same result as readline(), and iteration ends when the readline() method returns an empty string.

File objects also offer a number of other interesting attributes. These are not required for file-like objects, but should be implemented if they make sense for the particular object.

file.closed
: bool indicating the current state of the file object. This is a read-only attribute; the close() method changes the value. It may not be available on all file-like objects.

file.encoding
: The encoding that this file uses. When Unicode strings are written to a file, they will be converted to byte strings using this encoding. In addition, when the file is connected to a terminal, the attribute gives the encoding that the terminal is likely to use (that information might be incorrect if the user has misconfigured the terminal). The attribute is read-only and may not be present on all file-like objects. It may also be None, in which case the file uses the system default encoding for converting Unicode strings.

  New in version 2.3.

file.errors
: The Unicode error handler used along with the encoding.

  New in version 2.6.

file.mode
: The I/O mode for the file. If the file was created using the open() built-in function, this will be the value of the mode parameter. This is a read-only attribute and may not be present on all file-like objects.

file.name
: If the file object was created using open(), the name of the file. Otherwise, some string that indicates the source of the file object, of the form <...>. This is a read-only attribute and may not be present on all file-like objects.

file.newlines
: If Python was built with universal newlines enabled (the default) this read-only attribute exists, and for files opened in universal newline read mode it keeps track of the types of newlines encountered while reading the file. The values it can take are `'\r'`, `'\n'`, `'\r\n'`, None (unknown, no newlines

read yet) or a tuple containing all the newline types seen, to indicate that multiple newline conventions were encountered. For files not opened in universal newlines read mode the value of this attribute will be None.

file.softspace
: Boolean that indicates whether a space character needs to be printed before another value when using the print statement. Classes that are trying to simulate a file object should also have a writable softspace attribute, which should be initialized to zero. This will be automatic for most classes implemented in Python (care may be needed for objects that override attribute access); types implemented in C will have to provide a writable softspace attribute.

Note: This attribute is not used to control the print statement, but to allow the implementation of print to keep track of its internal state.

5.10 memoryview type

New in version 2.7.

memoryview objects allow Python code to access the internal data of an object that supports the buffer protocol without copying. Memory is generally interpreted as simple bytes.

class memoryview(obj)
: Create a memoryview that references obj. obj must support the buffer protocol. Built-in objects that support the buffer protocol include str and bytearray (but not unicode).

A memoryview has the notion of an element, which is the atomic memory unit handled by the originating object obj. For many simple types such as str and bytearray, an element is a single byte, but other third-party types may expose larger elements.

len(view) returns the total number of elements in the memoryview, view. The itemsize attribute will give you the number of bytes in a single element.

A memoryview supports slicing to expose its data. Taking a single index will return a single element as a str object. Full slicing will result in a subview:

```
>>> v = memoryview('abcefg')
>>> v[1]
'b'
>>> v[-1]
'g'
>>> v[1:4]
<memory at 0x77ab28>
>>> v[1:4].tobytes()
'bce'
```

If the object the memoryview is over supports changing its data, the memoryview supports slice assignment:

```
>>> data = bytearray('abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = 'z'
>>> data
bytearray(b'zbcefg')
```

(continues on next page)

(continued from previous page)

```
>>> v[1:4] = '123'
>>> data
bytearray(b'z123fg')
>>> v[2] = 'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot modify size of memoryview object
```

Notice how the size of the memoryview object cannot be changed.

memoryview has two methods:

tobytes()
: Return the data in the buffer as a bytestring (an object of class str).

```
>>> m = memoryview("abc")
>>> m.tobytes()
'abc'
```

tolist()
: Return the data in the buffer as a list of integers.

```
>>> memoryview("abc").tolist()
[97, 98, 99]
```

There are also several readonly attributes available:

format
: A string containing the format (in struct module style) for each element in the view. This defaults to `'B'`, a simple bytestring.

itemsize
: The size in bytes of each element of the memoryview.

shape
: A tuple of integers the length of ndim giving the shape of the memory as an N-dimensional array.

ndim
: An integer indicating how many dimensions of a multi-dimensional array the memory represents.

strides
: A tuple of integers the length of ndim giving the size in bytes to access each element for each dimension of the array.

readonly
: A bool indicating whether the memory is read only.

5.11 Context Manager Types

New in version 2.5.

Python's with statement supports the concept of a runtime context defined by a context manager. This is implemented using two separate methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends.

The context management protocol consists of a pair of methods that need to be provided for a context manager object to define a runtime context:

contextmanager.__enter__()
Enter the runtime context and return either this object or another object related to the runtime context. The value returned by this method is bound to the identifier in the as clause of with statements using this context manager.

An example of a context manager that returns itself is a file object. File objects return themselves from __enter__() to allow open() to be used as the context expression in a with statement.

An example of a context manager that returns a related object is the one returned by decimal.localcontext(). These managers set the active decimal context to a copy of the original decimal context and then return the copy. This allows changes to be made to the current decimal context in the body of the with statement without affecting code outside the with statement.

contextmanager.__exit__(exc_type, exc_val, exc_tb)
Exit the runtime context and return a Boolean flag indicating if any exception that occurred should be suppressed. If an exception occurred while executing the body of the with statement, the arguments contain the exception type, value and traceback information. Otherwise, all three arguments are None.

Returning a true value from this method will cause the with statement to suppress the exception and continue execution with the statement immediately following the with statement. Otherwise the exception continues propagating after this method has finished executing. Exceptions that occur during execution of this method will replace any exception that occurred in the body of the with statement.

The exception passed in should never be reraised explicitly - instead, this method should return a false value to indicate that the method completed successfully and does not want to suppress the raised exception. This allows context management code (such as contextlib.nested) to easily detect whether or not an __exit__() method has actually failed.

Python defines several context managers to support easy thread synchronisation, prompt closure of files or other objects, and simpler manipulation of the active decimal arithmetic context. The specific types are not treated specially beyond their implementation of the context management protocol. See the contextlib module for some examples.

Python's generators and the contextlib.contextmanager decorator provide a convenient way to implement these protocols. If a generator function is decorated with the contextlib.contextmanager decorator, it will return a context manager implementing the necessary __enter__() and __exit__() methods, rather than the iterator produced by an undecorated generator function.

Note that there is no specific slot for any of these methods in the type structure for Python objects in the Python/C API. Extension types wanting to define these methods must provide them as a normal Python accessible method. Compared to the overhead of setting up the runtime context, the overhead of a single class dictionary lookup is negligible.

5.12 Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

5.12.1 Modules

The only special operation on a module is attribute access: m.name, where m is a module and name accesses a name defined in m's symbol table. Module attributes can be assigned to. (Note that the import statement is not, strictly speaking, an operation on a module object; import foo does not require a module object named foo to exist, rather it requires an (external) definition for a module named foo somewhere.)

A special attribute of every module is __dict__. This is the dictionary containing the module's symbol table. Modifying this dictionary will actually change the module's symbol table, but direct assignment to

the __dict__ attribute is not possible (you can write m.__dict__['a'] = 1, which defines m.a to be 1, but you can't write m.__dict__ = {}). Modifying __dict__ directly is not recommended.

Modules built into the interpreter are written like this: <module 'sys' (built-in)>. If loaded from a file, they are written as <module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>.

5.12.2 Classes and Class Instances

See objects and class for these.

5.12.3 Functions

Function objects are created by function definitions. The only operation on a function object is to call it: func(argument-list).

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

See function for more information.

5.12.4 Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as append() on lists) and class instance methods. Built-in methods are described with the types that support them.

The implementation adds two special read-only attributes to class instance methods: m.im_self is the object on which the method operates, and m.im_func is the function implementing the method. Calling m(arg-1, arg-2, ..., arg-n) is completely equivalent to calling m.im_func(m.im_self, arg-1, arg-2, ..., arg-n).

Class instance methods are either bound or unbound, referring to whether the method was accessed through an instance or a class, respectively. When a method is unbound, its im_self attribute will be None and if called, an explicit self object must be passed as the first argument. In this case, self must be an instance of the unbound method's class (or a subclass of that class), otherwise a TypeError is raised.

Like function objects, methods objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (meth.im_func), setting method attributes on either bound or unbound methods is disallowed. Attempting to set an attribute on a method results in an AttributeError being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'instancemethod' object has no attribute 'whoami'
>>> c.method.im_func.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

See types for more information.

5.12.5 Code Objects

Code objects are used by the implementation to represent "pseudo-compiled" executable Python code such as a function body. They differ from function objects because they don't contain a reference to their global execution environment. Code objects are returned by the built-in compile() function and can be extracted from function objects through their func_code attribute. See also the code module.

A code object can be executed or evaluated by passing it (instead of a source string) to the exec statement or the built-in eval() function.

See types for more information.

5.12.6 Type Objects

Type objects represent the various object types. An object's type is accessed by the built-in function type(). There are no special operations on types. The standard module types defines names for all standard built-in types.

Types are written like this: <type 'int'>.

5.12.7 The Null Object

This object is returned by functions that don't explicitly return a value. It supports no special operations. There is exactly one null object, named None (a built-in name).

It is written as None.

5.12.8 The Ellipsis Object

This object is used by extended slice notation (see slicings). It supports no special operations. There is exactly one ellipsis object, named Ellipsis (a built-in name).

It is written as Ellipsis. When in a subscript, it can also be written as ..., for example seq[...].

5.12.9 The NotImplemented Object

This object is returned from comparisons and binary operations when they are asked to operate on types they don't support. See comparisons for more information.

It is written as NotImplemented.

5.12.10 Boolean Values

Boolean values are the two constant objects False and True. They are used to represent truth values (although other values can also be considered false or true). In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively. The built-in function bool() can be used to convert any value to a Boolean, if the value can be interpreted as a truth value (see section Truth Value Testing above).

They are written as False and True, respectively.

5.12.11 Internal Objects

See types for this information. It describes stack frame objects, traceback objects, and slice objects.

5.13 Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant. Some of these are not reported by the dir() built-in function.

object.__dict__
: A dictionary or other mapping object used to store an object's (writable) attributes.

object.__methods__
: Deprecated since version 2.2: Use the built-in function dir() to get a list of an object's attributes. This attribute is no longer available.

object.__members__
: Deprecated since version 2.2: Use the built-in function dir() to get a list of an object's attributes. This attribute is no longer available.

instance.__class__
: The class to which a class instance belongs.

class.__bases__
: The tuple of base classes of a class object.

definition.__name__
: The name of the class, type, function, method, descriptor, or generator instance.

The following attributes are only supported by new-style classes.

class.__mro__
: This attribute is a tuple of classes that are considered when looking for base classes during method resolution.

class.mro()
: This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in __mro__.

class.__subclasses__()
: Each new-style class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. Example:

```
>>> int.__subclasses__()
[<type 'bool'>]
```

CHAPTER

SIX

BUILT-IN EXCEPTIONS

Exceptions should be class objects. The exceptions are defined in the module exceptions. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace as well as the exceptions module.

For class exceptions, in a try statement with an except clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the raise statement. If the exception class is derived from the standard root class BaseException, the associated value is present as the exception instance's args attribute.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition "just like" the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the Exception class or one of its subclasses, and not from BaseException. More information on defining exceptions is available in the Python Tutorial under tut-userexceptions.

The following exceptions are only used as base classes for other exceptions.

exception BaseException
: The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use Exception). If str() or unicode() is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments.

  New in version 2.5.

  args
  : The tuple of arguments given to the exception constructor. Some built-in exceptions (like IOError) expect a certain number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

exception Exception
: All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

  Changed in version 2.5: Changed to inherit from BaseException.

exception StandardError
: The base class for all built-in exceptions except StopIteration, GeneratorExit, KeyboardInterrupt and SystemExit. StandardError itself is derived from Exception.

exception ArithmeticError
: The base class for those built-in exceptions that are raised for various arithmetic errors: OverflowError, ZeroDivisionError, FloatingPointError.

exception BufferError
: Raised when a buffer related operation cannot be performed.

exception LookupError
: The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: IndexError, KeyError. This can be raised directly by codecs.lookup().

exception EnvironmentError
: The base class for exceptions that can occur outside the Python system: IOError, OSError. When exceptions of this type are created with a 2-tuple, the first item is available on the instance's errno attribute (it is assumed to be an error number), and the second item is available on the strerror attribute (it is usually the associated error message). The tuple itself is also available on the args attribute.

  New in version 1.5.2.

  When an EnvironmentError exception is instantiated with a 3-tuple, the first two items are available as above, while the third item is available on the filename attribute. However, for backwards compatibility, the args attribute contains only a 2-tuple of the first two constructor arguments.

  The filename attribute is None when this exception is created with other than 3 arguments. The errno and strerror attributes are also None when the instance was created with other than 2 or 3 arguments. In this last case, args contains the verbatim constructor arguments as a tuple.

The following exceptions are the exceptions that are actually raised.

exception AssertionError
: Raised when an assert statement fails.

exception AttributeError
: Raised when an attribute reference (see attribute-references) or assignment fails. (When an object does not support attribute references or attribute assignments at all, TypeError is raised.)

exception EOFError
: Raised when one of the built-in functions (input() or raw_input()) hits an end-of-file condition (EOF) without reading any data. (N.B.: the file.read() and file.readline() methods return an empty string when they hit EOF.)

exception FloatingPointError
: Raised when a floating point operation fails. This exception is always defined, but can only be raised when Python is configured with the --with-fpectl option, or the WANT_SIGFPE_HANDLER symbol is defined in the pyconfig.h file.

exception GeneratorExit
: Raised when a generator's close() method is called. It directly inherits from BaseException instead of StandardError since it is technically not an error.

  New in version 2.5.

  Changed in version 2.6: Changed to inherit from BaseException.

exception IOError
: Raised when an I/O operation (such as a print statement, the built-in open() function or a method of a file object) fails for an I/O-related reason, e.g., "file not found" or "disk full".

  This class is derived from EnvironmentError. See the discussion above for more information on exception instance attributes.

Changed in version 2.6: Changed socket.error to use this as a base class.

exception ImportError
Raised when an import statement fails to find the module definition or when a from ... import fails to find a name that is to be imported.

exception IndexError
Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not a plain integer, TypeError is raised.)

exception KeyError
Raised when a mapping (dictionary) key is not found in the set of existing keys.

exception KeyboardInterrupt
Raised when the user hits the interrupt key (normally Control-C or Delete). During execution, a check for interrupts is made regularly. Interrupts typed when a built-in function input() or raw_input() is waiting for input also raise this exception. The exception inherits from BaseException so as to not be accidentally caught by code that catches Exception and thus prevent the interpreter from exiting.

Changed in version 2.5: Changed to inherit from BaseException.

exception MemoryError
Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C's malloc() function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

exception NameError
Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

exception NotImplementedError
This exception is derived from RuntimeError. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method.

New in version 1.5.2.

exception OSError
This exception is derived from EnvironmentError. It is raised when a function returns a system-related error (not for illegal argument types or other incidental errors). The errno attribute is a numeric error code from errno, and the strerror attribute is the corresponding string, as would be printed by the C function perror(). See the module errno, which contains names for the error codes defined by the underlying operating system.

For exceptions that involve a file system path (such as chdir() or unlink()), the exception instance will contain a third attribute, filename, which is the file name passed to the function.

New in version 1.5.2.

exception OverflowError
Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers (which would rather raise MemoryError than give up) and for most operations with plain integers, which return a long integer instead. Because of the lack of standardization of floating point exception handling in C, most floating point operations also aren't checked.

exception ReferenceError
This exception is raised when a weak reference proxy, created by the weakref.proxy() function, is used to access an attribute of the referent after it has been garbage collected. For more information on weak references, see the weakref module.

New in version 2.2: Previously known as the weakref.ReferenceError exception.

exception RuntimeError
Raised when an error is detected that doesn't fall in any of the other categories. The associated value is a string indicating what precisely went wrong.

exception StopIteration
Raised by an iterator's next() method to signal that there are no further values. This is derived from Exception rather than StandardError, since this is not considered an error in its normal application.

New in version 2.2.

exception SyntaxError
Raised when the parser encounters a syntax error. This may occur in an import statement, in an exec statement, in a call to the built-in function eval() or input(), or when reading the initial script or standard input (also interactively).

Instances of this class have attributes filename, lineno, offset and text for easier access to the details. str() of the exception instance returns only the message.

exception IndentationError
Base class for syntax errors related to incorrect indentation. This is a subclass of SyntaxError.

exception TabError
Raised when indentation contains an inconsistent use of tabs and spaces. This is a subclass of IndentationError.

exception SystemError
Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).

You should report this to the author or maintainer of your Python interpreter. Be sure to report the version of the Python interpreter (sys.version; it is also printed at the start of an interactive Python session), the exact error message (the exception's associated value) and if possible the source of the program that triggered the error.

exception SystemExit
This exception is raised by the sys.exit() function. When it is not handled, the Python interpreter exits; no stack traceback is printed. If the associated value is a plain integer, it specifies the system exit status (passed to C's exit() function); if it is None, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

Instances have an attribute code which is set to the proposed exit status or error message (defaulting to None). Also, this exception derives directly from BaseException and not StandardError, since it is not technically an error.

A call to sys.exit() is translated into an exception so that clean-up handlers (finally clauses of try statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The os._exit() function can be used if it is absolutely positively necessary to exit immediately (for example, in the child process after a call to os.fork()).

The exception inherits from BaseException instead of StandardError or Exception so that it is not accidentally caught by code that catches Exception. This allows the exception to properly propagate up and cause the interpreter to exit.

Changed in version 2.5: Changed to inherit from BaseException.

exception TypeError
Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

exception UnboundLocalError
Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of NameError.

New in version 2.0.

exception UnicodeError
Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of ValueError.

UnicodeError has attributes that describe the encoding or decoding error. For example, err.object[err.start:err.end] gives the particular invalid input that the codec failed on.

encoding
The name of the encoding that raised the error.

reason
A string describing the specific codec error.

object
The object the codec was attempting to encode or decode.

start
The first index of invalid data in object.

end
The index after the last invalid data in object.

New in version 2.0.

exception UnicodeEncodeError
Raised when a Unicode-related error occurs during encoding. It is a subclass of UnicodeError.

New in version 2.3.

exception UnicodeDecodeError
Raised when a Unicode-related error occurs during decoding. It is a subclass of UnicodeError.

New in version 2.3.

exception UnicodeTranslateError
Raised when a Unicode-related error occurs during translating. It is a subclass of UnicodeError.

New in version 2.3.

exception ValueError
Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as IndexError.

exception VMSError
Only available on VMS. Raised when a VMS-specific error occurs.

exception WindowsError
Raised when a Windows-specific error occurs or when the error number does not correspond to an errno value. The winerror and strerror values are created from the return values of the GetLastError() and FormatMessage() functions from the Windows Platform API. The errno value maps the winerror value to corresponding errno.h values. This is a subclass of OSError.

New in version 2.0.

Changed in version 2.5: Previous versions put the GetLastError() codes into errno.

exception ZeroDivisionError
Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

The following exceptions are used as warning categories; see the warnings module for more information.

exception Warning
: Base class for warning categories.

exception UserWarning
: Base class for warnings generated by user code.

exception DeprecationWarning
: Base class for warnings about deprecated features.

exception PendingDeprecationWarning
: Base class for warnings about features which will be deprecated in the future.

exception SyntaxWarning
: Base class for warnings about dubious syntax.

exception RuntimeWarning
: Base class for warnings about dubious runtime behavior.

exception FutureWarning
: Base class for warnings about constructs that will change semantically in the future.

exception ImportWarning
: Base class for warnings about probable mistakes in module imports.

  New in version 2.5.

exception UnicodeWarning
: Base class for warnings related to Unicode.

  New in version 2.5.

6.1 Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
```

(continues on next page)

(continued from previous page)

```
    |    |    +-- IndexError
    |    |    +-- KeyError
    |    +-- MemoryError
    |    +-- NameError
    |    |    +-- UnboundLocalError
    |    +-- ReferenceError
    |    +-- RuntimeError
    |    |    +-- NotImplementedError
    |    +-- SyntaxError
    |    |    +-- IndentationError
    |    |         +-- TabError
    |    +-- SystemError
    |    +-- TypeError
    |    +-- ValueError
    |         +-- UnicodeError
    |              +-- UnicodeDecodeError
    |              +-- UnicodeEncodeError
    |              +-- UnicodeTranslateError
    +-- Warning
         +-- DeprecationWarning
         +-- PendingDeprecationWarning
         +-- RuntimeWarning
         +-- SyntaxWarning
         +-- UserWarning
         +-- FutureWarning
         +-- ImportWarning
         +-- UnicodeWarning
         +-- BytesWarning
```

CHAPTER

SEVEN

STRING SERVICES

The modules described in this chapter provide a wide range of string manipulation operations.

In addition, Python's built-in string classes support the sequence type methods described in the Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange section, and also the string-specific methods described in the String Methods section. To output formatted strings use template strings or the % operator described in the String Formatting Operations section. Also, see the re module for string functions based on regular expressions.

7.1 string — Common string operations

Source code: Lib/string.py

The string module contains a number of useful constants and classes, as well as some deprecated legacy functions that are also available as methods on strings. In addition, Python's built-in string classes support the sequence type methods described in the Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange section, and also the string-specific methods described in the String Methods section. To output formatted strings use template strings or the % operator described in the String Formatting Operations section. Also, see the re module for string functions based on regular expressions.

7.1.1 String constants

The constants defined in this module are:

string.ascii_letters
: The concatenation of the ascii_lowercase and ascii_uppercase constants described below. This value is not locale-dependent.

string.ascii_lowercase
: The lowercase letters `'abcdefghijklmnopqrstuvwxyz'`. This value is not locale-dependent and will not change.

string.ascii_uppercase
: The uppercase letters `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. This value is not locale-dependent and will not change.

string.digits
: The string `'0123456789'`.

string.hexdigits
: The string `'0123456789abcdefABCDEF'`.

string.letters
: The concatenation of the strings lowercase and uppercase described below. The specific value is locale-dependent, and will be updated when locale.setlocale() is called.

string.lowercase
: A string containing all the characters that are considered lowercase letters. On most systems this is the string `'abcdefghijklmnopqrstuvwxyz'`. The specific value is locale-dependent, and will be updated when locale.setlocale() is called.

string.octdigits
: The string `'01234567'`.

string.punctuation
: String of ASCII characters which are considered punctuation characters in the C locale.

string.printable
: String of characters which are considered printable. This is a combination of digits, letters, punctuation, and whitespace.

string.uppercase
: A string containing all the characters that are considered uppercase letters. On most systems this is the string `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. The specific value is locale-dependent, and will be updated when locale.setlocale() is called.

string.whitespace
: A string containing all characters that are considered whitespace. On most systems this includes the characters space, tab, linefeed, return, formfeed, and vertical tab.

7.1.2 Custom String Formatting

New in version 2.6.

The built-in str and unicode classes provide the ability to do complex variable substitutions and value formatting via the str.format() method described in PEP 3101. The Formatter class in the string module allows you to create and customize your own string formatting behaviors using the same implementation as the built-in format() method.

class string.Formatter
: The Formatter class has the following public methods:

  format(format_string, *args, **kwargs)
  : The primary API method. It takes a format string and an arbitrary set of positional and keyword arguments. It is just a wrapper that calls vformat().

  vformat(format_string, args, kwargs)
  : This function does the actual work of formatting. It is exposed as a separate function for cases where you want to pass in a predefined dictionary of arguments, rather than unpacking and repacking the dictionary as individual arguments using the *args and **kwargs syntax. vformat() does the work of breaking up the format string into character data and replacement fields. It calls the various methods described below.

  In addition, the Formatter defines a number of methods that are intended to be replaced by subclasses:

  parse(format_string)
  : Loop over the format_string and return an iterable of tuples (literal_text, field_name, format_spec, conversion). This is used by vformat() to break the string into either literal text, or replacement fields.

The values in the tuple conceptually represent a span of literal text followed by a single replacement field. If there is no literal text (which can happen if two replacement fields occur consecutively), then literal_text will be a zero-length string. If there is no replacement field, then the values of field_name, format_spec and conversion will be None.

get_field(field_name, args, kwargs)
Given field_name as returned by parse() (see above), convert it to an object to be formatted. Returns a tuple (obj, used_key). The default version takes strings of the form defined in PEP 3101, such as "0[name]" or "label.title". args and kwargs are as passed in to vformat(). The return value used_key has the same meaning as the key parameter to get_value().

get_value(key, args, kwargs)
Retrieve a given field value. The key argument will be either an integer or a string. If it is an integer, it represents the index of the positional argument in args; if it is a string, then it represents a named argument in kwargs.

The args parameter is set to the list of positional arguments to vformat(), and the kwargs parameter is set to the dictionary of keyword arguments.

For compound field names, these functions are only called for the first component of the field name; Subsequent components are handled through normal attribute and indexing operations.

So for example, the field expression '0.name' would cause get_value() to be called with a key argument of 0. The name attribute will be looked up after get_value() returns by calling the built-in getattr() function.

If the index or keyword refers to an item that does not exist, then an IndexError or KeyError should be raised.

check_unused_args(used_args, args, kwargs)
Implement checking for unused arguments if desired. The arguments to this function is the set of all argument keys that were actually referred to in the format string (integers for positional arguments, and strings for named arguments), and a reference to the args and kwargs that was passed to vformat. The set of unused args can be calculated from these parameters. check_unused_args() is assumed to raise an exception if the check fails.

format_field(value, format_spec)
format_field() simply calls the global format() built-in. The method is provided so that subclasses can override it.

convert_field(value, conversion)
Converts the value (returned by get_field()) given a conversion type (as in the tuple returned by the parse() method). The default version understands 's' (str), 'r' (repr) and 'a' (ascii) conversion types.

7.1.3 Format String Syntax

The str.format() method and the Formatter class share the same syntax for format strings (although in the case of Formatter, subclasses can define their own format string syntax).

Format strings contain "replacement fields" surrounded by curly braces {}. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: {{ and }}.

The grammar for a replacement field is as follows:

```
replacement_field ::= "{" [field_name] ["!" conversion] [":" format_spec] "}"
```

```
field_name        ::=  arg_name ("." attribute_name | "[" element_index "]")*
arg_name          ::=  [identifier | integer]
attribute_name    ::=  identifier
element_index     ::=  integer | index_string
index_string      ::=  <any source character except "]"> +
conversion        ::=  "r" | "s"
format_spec       ::=  <described in the next section>
```

In less formal terms, the replacement field can start with a field_name that specifies the object whose value is to be formatted and inserted into the output instead of the replacement field. The field_name is optionally followed by a conversion field, which is preceded by an exclamation point `'!'`, and a format_spec, which is preceded by a colon `':'`. These specify a non-default format for the replacement value.

See also the Format Specification Mini-Language section.

The field_name itself begins with an arg_name that is either a number or a keyword. If it's a number, it refers to a positional argument, and if it's a keyword, it refers to a named keyword argument. If the numerical arg_names in a format string are 0, 1, 2, ... in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, ... will be automatically inserted in that order. Because arg_name is not quote-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings `'10'` or `':-]'`) within a format string. The arg_name can be followed by any number of index or attribute expressions. An expression of the form `'.name'` selects the named attribute using getattr(), while an expression of the form `'[index]'` does an index lookup using __getitem__().

Changed in version 2.7: The positional argument specifiers can be omitted for str.format() and unicode.format(), so `'{} {}'` is equivalent to `'{0} {1}'`, `u'{} {}'` is equivalent to `u'{0} {1}'`.

Some simple format string examples:

```
"First, thou shalt count to {0}"  # References first positional argument
"Bring me a {}"                   # Implicitly references the first positional argument
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"              # References keyword argument 'name'
"Weight in tons {0.weight}"       # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"   # First element of keyword argument 'players'.
```

The conversion field causes a type coercion before formatting. Normally, the job of formatting a value is done by the __format__() method of the value itself. However, in some cases it is desirable to force a type to be formatted as a string, overriding its own definition of formatting. By converting the value to a string before calling __format__(), the normal formatting logic is bypassed.

Two conversion flags are currently supported: `'!s'` which calls str() on the value, and `'!r'` which calls repr().

Some examples:

```
"Harold's a clever {0!s}"        # Calls str() on the argument first
"Bring out the holy {name!r}"    # Calls repr() on the argument first
```

The format_spec field contains a specification of how the value should be presented, including such details as field width, alignment, padding, decimal precision and so on. Each value type can define its own "formatting mini-language" or interpretation of the format_spec.

Most built-in types support a common formatting mini-language, which is described in the next section.

A format_spec field can also include nested replacement fields within it. These nested replacement fields may contain a field name, conversion flag and format specification, but deeper nesting is not allowed. The replacement fields within the format_spec are substituted before the format_spec string is interpreted. This allows the formatting of a value to be dynamically specified.

See the Format examples section for some examples.

Format Specification Mini-Language

"Format specifications" are used within replacement fields contained within a format string to define how individual values are presented (see Format String Syntax). They can also be passed directly to the built-in format() function. Each formattable type may define how the format specification is to be interpreted.

Most built-in types implement the following options for format specifications, although some of the formatting options are only supported by the numeric types.

A general convention is that an empty format string ("") produces the same result as if you had called str() on the value. A non-empty format string typically modifies the result.

The general form of a standard format specifier is:

```
format_spec     ::=  [[fill]align][sign][#][0][width][,][.precision][type]
fill            ::=  <any character>
align           ::=  "<" | ">" | "=" | "^"
sign            ::=  "+" | "-" | " "
width           ::=  integer
precision       ::=  integer
type            ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"
```

If a valid align value is specified, it can be preceded by a fill character that can be any character and defaults to a space if omitted. It is not possible to use a literal curly brace ("{" or "}") as the fill character when using the str.format() method. However, it is possible to insert a curly brace with a nested replacement field. This limitation doesn't affect the format() function.

The meaning of the various alignment options is as follows:

Option	Meaning
'<'	Forces the field to be left-aligned within the available space (this is the default for most objects).
'>'	Forces the field to be right-aligned within the available space (this is the default for numbers).
'='	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types. It becomes the default when '0' immediately precedes the field width.
'^'	Forces the field to be centered within the available space.

Note that unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.

The sign option is only valid for number types, and can be one of the following:

Option	Meaning
'+'	indicates that a sign should be used for both positive as well as negative numbers.
'-'	indicates that a sign should be used only for negative numbers (this is the default behavior).
space	indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

The '#' option is only valid for integers, and only for binary, octal, or hexadecimal output. If present, it specifies that the output will be prefixed by '0b', '0o', or '0x', respectively.

The ',' option signals the use of a comma for a thousands separator. For a locale aware separator, use the 'n' integer presentation type instead.

Changed in version 2.7: Added the ',' option (see also PEP 378).

width is a decimal integer defining the minimum field width. If not specified, then the field width will be determined by the content.

When no explicit alignment is given, preceding the width field by a zero ('0') character enables sign-aware zero-padding for numeric types. This is equivalent to a fill character of '0' with an alignment type of '='.

The precision is a decimal number indicating how many digits should be displayed after the decimal point for a floating point value formatted with 'f' and 'F', or before and after the decimal point for a floating point value formatted with 'g' or 'G'. For non-number types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The precision is not allowed for integer values.

Finally, the type determines how the data should be presented.

The available string presentation types are:

Type	Meaning
's'	String format. This is the default type for strings and may be omitted.
None	The same as 's'.

The available integer presentation types are:

Type	Meaning
'b'	Binary format. Outputs the number in base 2.
'c'	Character. Converts the integer to the corresponding unicode character before printing.
'd'	Decimal Integer. Outputs the number in base 10.
'o'	Octal format. Outputs the number in base 8.
'x'	Hex format. Outputs the number in base 16, using lower- case letters for the digits above 9.
'X'	Hex format. Outputs the number in base 16, using upper- case letters for the digits above 9.
'n'	Number. This is the same as 'd', except that it uses the current locale setting to insert the appropriate number separator characters.
None	The same as 'd'.

In addition to the above presentation types, integers can be formatted with the floating point presentation types listed below (except 'n' and None). When doing so, float() is used to convert the integer to a floating point number before formatting.

The available presentation types for floating point and decimal values are:

Type	Meaning
'e'	Exponent notation. Prints the number in scientific notation using the letter 'e' to indicate the exponent. The default precision is 6.
'E'	Exponent notation. Same as 'e' except it uses an upper case 'E' as the separator character.
'f'	Fixed-point notation. Displays the number as a fixed-point number. The default precision is 6.
'F'	Fixed point notation. Same as 'f'.
'g'	General format. For a given precision p >= 1, this rounds the number to p significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. The precise rules are as follows: suppose that the result formatted with presentation type 'e' and precision p-1 would have exponent exp. Then if -4 <= exp < p, the number is formatted with presentation type 'f' and precision p-1-exp. Otherwise, the number is formatted with presentation type 'e' and precision p-1. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it. Positive and negative infinity, positive and negative zero, and nans, are formatted as inf, -inf, 0, -0 and nan respectively, regardless of the precision. A precision of 0 is treated as equivalent to a precision of 1. The default precision is 6.
'G'	General format. Same as 'g' except switches to 'E' if the number gets too large. The representations of infinity and NaN are uppercased, too.
'n'	Number. This is the same as 'g', except that it uses the current locale setting to insert the appropriate number separator characters.
'%'	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.
None	The same as 'g'.

Format examples

This section contains examples of the str.format() syntax and comparison with the old %-formatting.

In most of the cases the syntax is similar to the old %-formatting, with the addition of the {} and with : used instead of %. For example, '%03.2f' can be translated to '{:03.2f}'.

The new format syntax also supports new and different options, shown in the following examples.

Accessing arguments by position:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 2.7+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'abracadabra'
```

Accessing arguments by name:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordinates: 37.24N, -115.81W'
```

(continues on next page)

(continued from previous page)

```
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Accessing arguments' attributes:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.0.'
>>> class Point(object):
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

Accessing arguments' items:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

Replacing %s and %r:

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

Aligning the text and specifying a width:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

Replacing %+f, %-f, and % f and specifying a sign:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

Replacing %x and %o and converting the value to different bases:

```
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
```

(continues on next page)

(continued from previous page)

```
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

Using the comma as a thousands separator:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

Expressing a percentage:

```
>>> points = 19.5
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 88.64%'
```

Using type-specific formatting:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

Nesting arguments and more complex examples:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print '{0:{width}{base}}'.format(num, base=base, width=width),
...     print
...
    5     5     5   101
    6     6     6   110
    7     7     7   111
    8     8    10  1000
    9     9    11  1001
   10     A    12  1010
   11     B    13  1011
```

7.1.4 Template strings

New in version 2.4.

Templates provide simpler string substitutions as described in PEP 292. Instead of the normal %-based substitutions, Templates support $-based substitutions, using the following rules:

- $$ is an escape; it is replaced with a single $.
- $identifier names a substitution placeholder matching a mapping key of "identifier". By default, "identifier" must spell a Python identifier. The first non-identifier character after the $ character terminates this placeholder specification.
- ${identifier} is equivalent to $identifier. It is required when valid identifier characters follow the placeholder but are not part of the placeholder, such as "${noun}ification".

Any other appearance of $ in the string will result in a ValueError being raised.

The string module provides a Template class that implements these rules. The methods of Template are:

class string.Template(template)
: The constructor takes a single argument which is the template string.

  substitute(mapping[, **kws])
  : Performs the template substitution, returning a new string. mapping is any dictionary-like object with keys that match the placeholders in the template. Alternatively, you can provide keyword arguments, where the keywords are the placeholders. When both mapping and kws are given and there are duplicates, the placeholders from kws take precedence.

  safe_substitute(mapping[, **kws])
  : Like substitute(), except that if placeholders are missing from mapping and kws, instead of raising a KeyError exception, the original placeholder will appear in the resulting string intact. Also, unlike with substitute(), any other appearances of the $ will simply return $ instead of raising ValueError.

    While other exceptions may still occur, this method is called "safe" because it always tries to return a usable string instead of raising an exception. In another sense, safe_substitute() may be anything other than safe, since it will silently ignore malformed templates containing dangling delimiters, unmatched braces, or placeholders that are not valid Python identifiers.

  Template instances also provide one public data attribute:

  template
  : This is the object passed to the constructor's template argument. In general, you shouldn't change it, but read-only access is not enforced.

Here is an example of how to use a Template:

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

Advanced usage: you can derive subclasses of Template to customize the placeholder syntax, delimiter character, or the entire regular expression used to parse template strings. To do this, you can override these class attributes:

- delimiter – This is the literal string describing a placeholder introducing delimiter. The default value is $. Note that this should not be a regular expression, as the implementation will call re.escape() on this string as needed.
- idpattern – This is the regular expression describing the pattern for non-braced placeholders (the braces will be added automatically as appropriate). The default value is the regular expression [_a-z][_a-z0-9]*.

Alternatively, you can provide the entire regular expression pattern by overriding the class attribute pattern. If you do this, the value must be a regular expression object with four named capturing groups. The capturing groups correspond to the rules given above, along with the invalid placeholder rule:

- escaped – This group matches the escape sequence, e.g. $$, in the default pattern.
- named – This group matches the unbraced placeholder name; it should not include the delimiter in capturing group.
- braced – This group matches the brace enclosed placeholder name; it should not include either the delimiter or braces in the capturing group.
- invalid – This group matches any other delimiter pattern (usually a single delimiter), and it should appear last in the regular expression.

7.1.5 String functions

The following functions are available to operate on string and Unicode objects. They are not available as string methods.

string.capwords(s[, sep])
: Split the argument into words using str.split(), capitalize each word using str.capitalize(), and join the capitalized words using str.join(). If the optional second argument sep is absent or None, runs of whitespace characters are replaced by a single space and leading and trailing whitespace are removed, otherwise sep is used to split and join the words.

string.maketrans(from, to)
: Return a translation table suitable for passing to translate(), that will map each character in from into the character at the same position in to; from and to must have the same length.

Note: Don't use strings derived from lowercase and uppercase as arguments; in some locales, these don't have the same length. For case conversions, always use str.lower() and str.upper().

7.1.6 Deprecated string functions

The following list of functions are also defined as methods of string and Unicode objects; see section String Methods for more information on those. You should consider these functions as deprecated, although they will not be removed until Python 3. The functions defined in this module are:

string.atof(s)
: Deprecated since version 2.0: Use the float() built-in function.

Convert a string to a floating point number. The string must have the standard syntax for a floating point literal in Python, optionally preceded by a sign (+ or -). Note that this behaves identical to the built-in function float() when passed a string.

Note: When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. The specific set of strings accepted which cause these values to be returned depends entirely on the C library and is known to vary.

---

string.atoi(s[, base])
Deprecated since version 2.0: Use the int() built-in function.

Convert string s to an integer in the given base. The string must consist of one or more digits, optionally preceded by a sign (+ or -). The base defaults to 10. If it is 0, a default base is chosen depending on the leading characters of the string (after stripping the sign): 0x or 0X means 16, 0 means 8, anything else means 10. If base is 16, a leading 0x or 0X is always accepted, though not required. This behaves identically to the built-in function int() when passed a string. (Also note: for a more flexible interpretation of numeric literals, use the built-in function eval().)

string.atol(s[, base])
Deprecated since version 2.0: Use the long() built-in function.

Convert string s to a long integer in the given base. The string must consist of one or more digits, optionally preceded by a sign (+ or -). The base argument has the same meaning as for atoi(). A trailing l or L is not allowed, except if the base is 0. Note that when invoked without base or with base set to 10, this behaves identical to the built-in function long() when passed a string.

string.capitalize(word)
Return a copy of word with only its first character capitalized.

string.expandtabs(s[, tabsize])
Expand tabs in a string replacing them by one or more spaces, depending on the current column and the given tab size. The column number is reset to zero after each newline occurring in the string. This doesn't understand other non-printing characters or escape sequences. The tab size defaults to 8.

string.find(s, sub[, start[, end]])
Return the lowest index in s where the substring sub is found such that sub is wholly contained in s[start:end]. Return -1 on failure. Defaults for start and end and interpretation of negative values is the same as for slices.

string.rfind(s, sub[, start[, end]])
Like find() but find the highest index.

string.index(s, sub[, start[, end]])
Like find() but raise ValueError when the substring is not found.

string.rindex(s, sub[, start[, end]])
Like rfind() but raise ValueError when the substring is not found.

string.count(s, sub[, start[, end]])
Return the number of (non-overlapping) occurrences of substring sub in string s[start:end]. Defaults for start and end and interpretation of negative values are the same as for slices.

string.lower(s)
Return a copy of s, but with upper case letters converted to lower case.

string.split(s[, sep[, maxsplit]])
Return a list of the words of the string s. If the optional second argument sep is absent or None, the words are separated by arbitrary strings of whitespace characters (space, tab, newline, return,

formfeed). If the second argument sep is present and not None, it specifies a string to be used as the word separator. The returned list will then have one more item than the number of non-overlapping occurrences of the separator in the string. If maxsplit is given, at most maxsplit number of splits occur, and the remainder of the string is returned as the final element of the list (thus, the list will have at most maxsplit+1 elements). If maxsplit is not specified or -1, then there is no limit on the number of splits (all possible splits are made).

The behavior of split on an empty string depends on the value of sep. If sep is not specified, or specified as None, the result will be an empty list. If sep is specified as any string, the result will be a list containing one element which is an empty string.

string.rsplit(s[, sep[, maxsplit]])
Return a list of the words of the string s, scanning s from the end. To all intents and purposes, the resulting list of words is the same as returned by split(), except when the optional third argument maxsplit is explicitly specified and nonzero. If maxsplit is given, at most maxsplit number of splits – the rightmost ones – occur, and the remainder of the string is returned as the first element of the list (thus, the list will have at most maxsplit+1 elements).

New in version 2.4.

string.splitfields(s[, sep[, maxsplit]])
This function behaves identically to split(). (In the past, split() was only used with one argument, while splitfields() was only used with two arguments.)

string.join(words[, sep])
Concatenate a list or tuple of words with intervening occurrences of sep. The default value for sep is a single space character. It is always true that string.join(string.split(s, sep), sep) equals s.

string.joinfields(words[, sep])
This function behaves identically to join(). (In the past, join() was only used with one argument, while joinfields() was only used with two arguments.) Note that there is no joinfields() method on string objects; use the join() method instead.

string.lstrip(s[, chars])
Return a copy of the string with leading characters removed. If chars is omitted or None, whitespace characters are removed. If given and not None, chars must be a string; the characters in the string will be stripped from the beginning of the string this method is called on.

Changed in version 2.2.3: The chars parameter was added. The chars parameter cannot be passed in earlier 2.2 versions.

string.rstrip(s[, chars])
Return a copy of the string with trailing characters removed. If chars is omitted or None, whitespace characters are removed. If given and not None, chars must be a string; the characters in the string will be stripped from the end of the string this method is called on.

Changed in version 2.2.3: The chars parameter was added. The chars parameter cannot be passed in earlier 2.2 versions.

string.strip(s[, chars])
Return a copy of the string with leading and trailing characters removed. If chars is omitted or None, whitespace characters are removed. If given and not None, chars must be a string; the characters in the string will be stripped from the both ends of the string this method is called on.

Changed in version 2.2.3: The chars parameter was added. The chars parameter cannot be passed in earlier 2.2 versions.

string.swapcase(s)
Return a copy of s, but with lower case letters converted to upper case and vice versa.

string.translate(s, table[, deletechars])
: Delete all characters from s that are in deletechars (if present), and then translate the characters using table, which must be a 256-character string giving the translation for each character value, indexed by its ordinal. If table is None, then only the character deletion step is performed.

string.upper(s)
: Return a copy of s, but with lower case letters converted to upper case.

string.ljust(s, width[, fillchar])
string.rjust(s, width[, fillchar])
string.center(s, width[, fillchar])
: These functions respectively left-justify, right-justify and center a string in a field of given width. They return a string that is at least width characters wide, created by padding the string s with the character fillchar (default is a space) until the given width on the right, left or both sides. The string is never truncated.

string.zfill(s, width)
: Pad a numeric string s on the left with zero digits until the given width is reached. Strings starting with a sign are handled correctly.

string.replace(s, old, new[, maxreplace])
: Return a copy of string s with all occurrences of substring old replaced by new. If the optional argument maxreplace is given, the first maxreplace occurrences are replaced.

7.2 re — Regular expression operations

This module provides regular expression matching operations similar to those found in Perl. Both patterns and strings to be searched can be Unicode strings as well as 8-bit strings.

Regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python's usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write `'\\\\'` as the pattern string, because the regular expression must be \\, and each backslash must be expressed as \\ inside a regular Python string literal.

The solution is to use Python's raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with `'r'`. So `r"\n"` is a two-character string containing `'\'` and `'n'`, while `"\n"` is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

It is important to note that most regular expression operations are available as module-level functions and RegexObject methods. The functions are shortcuts that don't require you to compile a regex object first, but miss some fine-tuning parameters.

See also:

The third-party regex module, which has an API compatible with the standard library re module, but offers additional functionality and a more thorough Unicode support.

7.2.1 Regular Expression Syntax

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Regular expressions can be concatenated to form new regular expressions; if A and B are both regular expressions, then AB is also a regular expression. In general, if a string p matches A and another string q matches B, the string pq will match AB. This holds unless A or B contain low precedence operations; boundary conditions between A and B; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book referenced above, or almost any textbook about compiler construction.

A brief explanation of the format of regular expressions follows. For further information and a gentler presentation, consult the regex-howto.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like `'A'`, `'a'`, or `'0'`, are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so last matches the string `'last'`. (In the rest of this section, we'll write RE's in this special style, usually without quotes, and strings to be matched `'`in single quotes`'`.)

Some characters, like `'|'` or `'('`, are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted. Regular expression pattern strings may not contain null bytes, but can specify the null byte using the \number notation, e.g., `'\x00'`.

Repetition qualifiers (*, +, ?, {m,n}, etc) cannot be directly nested. This avoids ambiguity with the non-greedy modifier suffix ?, and with other modifiers in other implementations. To apply a second repetition to an inner repetition, parentheses may be used. For example, the expression (?:a{6})* matches any multiple of six `'a'` characters.

The special characters are:

`'.'` (Dot.) In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline.

`'^'` (Caret.) Matches the start of the string, and in MULTILINE mode also matches immediately after each newline.

`'$'` Matches the end of the string or just before the newline at the end of the string, and in MULTILINE mode also matches before a newline. foo matches both 'foo' and 'foobar', while the regular expression foo$ matches only 'foo'. More interestingly, searching for foo.$ in `'foo1\nfoo2\n'` matches 'foo2' normally, but 'foo1' in MULTILINE mode; searching for a single $ in `'foo\n'` will find two (empty) matches: one just before the newline, and one at the end of the string.

`'*'` Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ab* will match 'a', 'ab', or 'a' followed by any number of 'b's.

`'+'` Causes the resulting RE to match 1 or more repetitions of the preceding RE. ab+ will match 'a' followed by any non-zero number of 'b's; it will not match just 'a'.

`'?'` Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. ab? will match either 'a' or 'ab'.

?, +?, ?? The `'*'`, `'+'`, and `'?'` qualifiers are all greedy; they match as much text as possible. Sometimes this behaviour isn't desired; if the RE <.> is matched against <a> b <c>, it will match the entire string, and not just <a>. Adding ? after the qualifier makes it perform the match in non-greedy or minimal fashion; as few characters as possible will be matched. Using the RE <.*?> will match only <a>.

{m} Specifies that exactly m copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, a{6} will match exactly six `'a'` characters, but not five.

{m,n} Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, a{3,5} will match from 3 to 5 `'a'` characters. Omitting m specifies a lower bound of zero, and omitting n specifies an infinite upper bound. As an example,

a{4,}b will match aaaab or a thousand 'a' characters followed by a b, but not aaab. The comma may not be omitted or the modifier would be confused with the previously described form.

{m,n}? Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the non-greedy version of the previous qualifier. For example, on the 6-character string 'aaaaaa', a{3,5} will match 5 'a' characters, while a{3,5}? will only match 3 characters.

'\' Either escapes special characters (permitting you to match characters like '*', '?', and so forth), or signals a special sequence; special sequences are discussed below.

If you're not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn't recognized by Python's parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it's highly recommended that you use raw strings for all but the simplest expressions.

[] Used to indicate a set of characters. In a set:

- Characters can be listed individually, e.g. [amk] will match 'a', 'm', or 'k'.
- Ranges of characters can be indicated by giving two characters and separating them by a '-', for example [a-z] will match any lowercase ASCII letter, [0-5][0-9] will match all the two-digits numbers from 00 to 59, and [0-9A-Fa-f] will match any hexadecimal digit. If - is escaped (e.g. [a\-z]) or if it's placed as the first or last character (e.g. [a-]), it will match a literal '-'.
- Special characters lose their special meaning inside sets. For example, [(+*)] will match any of the literal characters '(', '+', '*', or ')'.
- Character classes such as \w or \S (defined below) are also accepted inside a set, although the characters they match depends on whether LOCALE or UNICODE mode is in force.
- Characters that are not within a range can be matched by complementing the set. If the first character of the set is '^', all the characters that are not in the set will be matched. For example, [^5] will match any character except '5', and [^^] will match any character except '^'. ^ has no special meaning if it's not the first character in the set.
- To match a literal ']' inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both [()[\]{}] and []()[{}] will both match a parenthesis.

'|' A|B, where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the '|' in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by '|' are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once A matches, B will not be tested further, even if it would produce a longer overall match. In other words, the '|' operator is never greedy. To match a literal '|', use \|, or enclose it inside a character class, as in [|].

(...) Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the \number special sequence, described below. To match the literals '(' or ')', use \(or \), or enclose them inside a character class: [(] [)].

(?...) This is an extension notation (a '?' following a '(' is not meaningful otherwise). The first character after the '?' determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; (?P<name>...) is the only exception to this rule. Following are the currently supported extensions.

(?iLmsux) (One or more letters from the set 'i', 'L', 'm', 's', 'u', 'x'.) The group matches the empty string; the letters set the corresponding flags: re.I (ignore case), re.L (locale dependent), re.M (multi-line), re.S (dot matches all), re.U (Unicode dependent), and re.X (verbose), for the entire regular

expression. (The flags are described in Module Contents.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a flag argument to the re.compile() function.

Note that the (?x) flag changes how the expression is parsed. It should be used first in the expression string, or after one or more whitespace characters. If there are non-whitespace characters before the flag, the results are undefined.

(?:...) A non-capturing version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group cannot be retrieved after performing a match or referenced later in the pattern.

(?P<name>...) Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name name. Group names must be valid Python identifiers, and each group name must be defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named.

Named groups can be referenced in three contexts. If the pattern is (?P<quote>['"]).*?(?P=quote) (i.e. matching a string quoted with either single or double quotes):

Context of reference to group "quote"	Ways to reference it
in the same pattern itself	• (?P=quote) (as shown) • \1
when processing match object m	• m.group('quote') • m.end('quote') (etc.)
in a string passed to the repl argument of re.sub()	• \g<quote> • \g<1> • \1

(?P=name) A backreference to a named group; it matches whatever text was matched by the earlier group named name.

(?#...) A comment; the contents of the parentheses are simply ignored.

(?=...) Matches if ... matches next, but doesn't consume any of the string. This is called a lookahead assertion. For example, Isaac (?=Asimov) will match 'Isaac ' only if it's followed by 'Asimov'.

(?!...) Matches if ... doesn't match next. This is a negative lookahead assertion. For example, Isaac (?!Asimov) will match 'Isaac ' only if it's not followed by 'Asimov'.

(?<=...) Matches if the current position in the string is preceded by a match for ... that ends at the current position. This is called a positive lookbehind assertion. (?<=abc)def will find a match in abcdef, since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that abc or a|b are allowed, but a* and a{3,4} are not. Group references are not supported even if they match strings of some fixed length. Note that patterns which start with positive lookbehind assertions will not match at the beginning of the string being searched; you will most likely want to use the search() function rather than the match() function:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

This example looks for a word following a hyphen:

```
>>> m = re.search('(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

(?<!...) Matches if the current position in the string is not preceded by a match for .... This is called a negative lookbehind assertion. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length and shouldn't contain group references. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

(?(id/name)yes-pattern|no-pattern) Will try to match with yes-pattern if the group with given id or name exists, and with no-pattern if it doesn't. no-pattern is optional and can be omitted. For example, (<)?(\w+@\w+(?:\.\w+)+)(?(1)>) is a poor email matching pattern, which will match with '<user@host.com>' as well as 'user@host.com', but not with '<user@host.com'.

New in version 2.4.

The special sequences consist of '\' and a character from the list below. If the ordinary character is not on the list, then the resulting RE will match the second character. For example, \$ matches the character '$'.

\number Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, (.+) \1 matches 'the the' or '55 55', but not 'thethe' (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of number is 0, or number is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value number. Inside the '[' and ']' of a character class, all numeric escapes are treated as characters.

\A Matches only at the start of the string.

\b Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of alphanumeric or underscore characters, so the end of a word is indicated by whitespace or a non-alphanumeric, non-underscore character. Note that formally, \b is defined as the boundary between a \w and a \W character (or vice versa), or between \w and the beginning/end of the string, so the precise set of characters deemed to be alphanumeric depends on the values of the UNICODE and LOCALE flags. For example, r'\bfoo\b' matches 'foo', 'foo.', '(foo)', 'bar foo baz' but not 'foobar' or 'foo3'. Inside a character range, \b represents the backspace character, for compatibility with Python's string literals.

\B Matches the empty string, but only when it is not at the beginning or end of a word. This means that r'py\B' matches 'python', 'py3', 'py2', but not 'py', 'py.', or 'py!'. \B is just the opposite of \b, so is also subject to the settings of LOCALE and UNICODE.

\d When the UNICODE flag is not specified, matches any decimal digit; this is equivalent to the set [0-9]. With UNICODE, it will match whatever is classified as a decimal digit in the Unicode character properties database.

\D When the UNICODE flag is not specified, matches any non-digit character; this is equivalent to the set [^0-9]. With UNICODE, it will match anything other than character marked as digits in the Unicode character properties database.

\s When the UNICODE flag is not specified, it matches any whitespace character, this is equivalent to the set [\t\n\r\f\v]. The LOCALE flag has no extra effect on matching of the space. If UNICODE is set, this will match the characters [\t\n\r\f\v] plus whatever is classified as space in the Unicode character properties database.

\S When the UNICODE flag is not specified, matches any non-whitespace character; this is equivalent to the set [^ \t\n\r\f\v] The LOCALE flag has no extra effect on non-whitespace match. If UNICODE is set, then any character not marked as space in the Unicode character properties database is matched.

\w When the LOCALE and UNICODE flags are not specified, matches any alphanumeric character and the underscore; this is equivalent to the set [a-zA-Z0-9_]. With LOCALE, it will match the set [0-9_] plus whatever characters are defined as alphanumeric for the current locale. If UNICODE is set, this will match the characters [0-9_] plus whatever is classified as alphanumeric in the Unicode character properties database.

\W When the LOCALE and UNICODE flags are not specified, matches any non-alphanumeric character; this is equivalent to the set [^a-zA-Z0-9_]. With LOCALE, it will match any character not in the set [0-9_], and not defined as alphanumeric for the current locale. If UNICODE is set, this will match anything other than [0-9_] plus characters classified as not alphanumeric in the Unicode character properties database.

\Z Matches only at the end of the string.

If both LOCALE and UNICODE flags are included for a particular sequence, then LOCALE flag takes effect first followed by the UNICODE.

Most of the standard escapes supported by Python string literals are also accepted by the regular expression parser:

```
\a      \b      \f      \n
\r      \t      \v      \x
\\
```

(Note that \b is used to represent word boundaries, and means "backspace" only inside character classes.)

Octal escapes are included in a limited form: If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. Otherwise, it is a group reference. As for string literals, octal escapes are always at most three digits in length.

See also:

Mastering Regular Expressions Book on regular expressions by Jeffrey Friedl, published by O'Reilly. The second edition of the book no longer covers Python at all, but the first edition covered writing good regular expression patterns in great detail.

7.2.2 Module Contents

The module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions. Most non-trivial applications always use the compiled form.

re.compile(pattern, flags=0)

Compile a regular expression pattern into a regular expression object, which can be used for matching using its match() and search() methods, described below.

The expression's behaviour can be modified by specifying a flags value. Values can be any of the following variables, combined using bitwise OR (the | operator).

The sequence

```
prog = re.compile(pattern)
result = prog.match(string)
```

is equivalent to

```
result = re.match(pattern, string)
```

but using re.compile() and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

Note: The compiled versions of the most recent patterns passed to re.match(), re.search() or re.compile() are cached, so programs that use only a few regular expressions at a time needn't worry about compiling regular expressions.

---

re.DEBUG
Display debug information about compiled expression.

re.I
re.IGNORECASE
Perform case-insensitive matching; expressions like [A-Z] will match lowercase letters, too. This is not affected by the current locale. To get this effect on non-ASCII Unicode characters such as ü and Ü, add the UNICODE flag.

re.L
re.LOCALE
Make \w, \W, \b, \B, \s and \S dependent on the current locale.

re.M
re.MULTILINE
When specified, the pattern character '^' matches at the beginning of the string and at the beginning of each line (immediately following each newline); and the pattern character '$' matches at the end of the string and at the end of each line (immediately preceding each newline). By default, '^' matches only at the beginning of the string, and '$' only at the end of the string and immediately before the newline (if any) at the end of the string.

re.S
re.DOTALL
Make the '.' special character match any character at all, including a newline; without this flag, '.' will match anything except a newline.

re.U
re.UNICODE
Make the \w, \W, \b, \B, \d, \D, \s and \S sequences dependent on the Unicode character properties database. Also enables non-ASCII matching for IGNORECASE.

New in version 2.0.

re.X
re.VERBOSE
This flag allows you to write regular expressions that look nicer and are more readable by allowing you to visually separate logical sections of the pattern and add comments. Whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash, or within tokens like *?, (?: or (?P<...>. When a line contains a # that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such # through the end of the line are ignored.

This means that the two following regular expression objects that match a decimal number are functionally equal:

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

re.search(pattern, string, flags=0)
: Scan through string looking for the first location where the regular expression pattern produces a match, and return a corresponding MatchObject instance. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

re.match(pattern, string, flags=0)
: If zero or more characters at the beginning of string match the regular expression pattern, return a corresponding MatchObject instance. Return None if the string does not match the pattern; note that this is different from a zero-length match.

  Note that even in MULTILINE mode, re.match() will only match at the beginning of the string and not at the beginning of each line.

  If you want to locate a match anywhere in string, use search() instead (see also search() vs. match()).

re.split(pattern, string, maxsplit=0, flags=0)
: Split string by the occurrences of pattern. If capturing parentheses are used in pattern, then the text of all groups in the pattern are also returned as part of the resulting list. If maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list. (Incompatibility note: in the original Python 1.5 release, maxsplit was ignored. This has been fixed in later releases.)

```
>>> re.split('\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split('\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

  If there are capturing groups in the separator and it matches at the start of the string, the result will start with an empty string. The same holds for the end of the string:

```
>>> re.split('(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

  That way, separator components are always found at the same relative indices within the result list (e.g., if there's one capturing group in the separator, the 0th, the 2nd and so forth).

  Note that split will never split a string on an empty pattern match. For example:

```
>>> re.split('x*', 'foo')
['foo']
>>> re.split("(?m)^$", "foo\n\nbar\n")
['foo\n\nbar\n']
```

  Changed in version 2.7: Added the optional flags argument.

re.findall(pattern, string, flags=0)
: Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result.

Note: Due to the limitation of the current implementation the character following an empty match is not included in a next match, so findall(r'^|\w+', 'two words') returns ['', 'wo', 'words'] (note

missed "t"). This is changed in Python 3.7.

New in version 1.5.2.

Changed in version 2.4: Added the optional flags argument.

re.finditer(pattern, string, flags=0)
Return an iterator yielding MatchObject instances over all non-overlapping matches for the RE pattern in string. The string is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result. See also the note about findall().

New in version 2.2.

Changed in version 2.4: Added the optional flags argument.

re.sub(pattern, repl, string, count=0, flags=0)
Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl. If the pattern isn't found, string is returned unchanged. repl can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, \n is converted to a single newline character, \r is converted to a carriage return, and so forth. Unknown escapes such as \j are left alone. Backreferences, such as \6, are replaced with the substring matched by group 6 in the pattern. For example:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\npy_\1(void)\n{',
...        'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

If repl is a function, it is called for every non-overlapping occurrence of pattern. The function takes a single match object argument, and returns the replacement string. For example:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

The pattern may be a string or an RE object.

The optional argument count is the maximum number of pattern occurrences to be replaced; count must be a non-negative integer. If omitted or zero, all occurrences will be replaced. Empty matches for the pattern are replaced only when not adjacent to a previous match, so sub('x*', '-', 'abc') returns '-a-b-c-'.

In string-type repl arguments, in addition to the character escapes and backreferences described above, \g<name> will use the substring matched by the group named name, as defined by the (?P<name>...) syntax. \g<number> uses the corresponding group number; \g<2> is therefore equivalent to \2, but isn't ambiguous in a replacement such as \g<2>0. \20 would be interpreted as a reference to group 20, not a reference to group 2 followed by the literal character '0'. The backreference \g<0> substitutes in the entire substring matched by the RE.

Changed in version 2.7: Added the optional flags argument.

re.subn(pattern, repl, string, count=0, flags=0)
Perform the same operation as sub(), but return a tuple (new_string, number_of_subs_made).

Changed in version 2.7: Added the optional flags argument.

re.escape(pattern)
Escape all the characters in pattern except ASCII letters and numbers. This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it. For example:

```
>>> print re.escape('python.exe')
python\.exe

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print '[%s]+' % re.escape(legal_chars)
[abcdefghijklmnopqrstuvwxyz0123456789\!\#\$\%\&\'\*\+\-\.\^\_\`\|\~\:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print '|'.join(map(re.escape, sorted(operators, reverse=True)))
\/|\-|\+|\*\*|\*
```

re.purge()
Clear the regular expression cache.

exception re.error
Exception raised when a string passed to one of the functions here is not a valid regular expression (for example, it might contain unmatched parentheses) or when some other error occurs during compilation or matching. It is never an error if a string contains no match for a pattern.

7.2.3 Regular Expression Objects

class re.RegexObject
The RegexObject class supports the following methods and attributes:

search(string[, pos[, endpos]])
Scan through string looking for a location where this regular expression produces a match, and return a corresponding MatchObject instance. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The optional second parameter pos gives an index in the string where the search is to start; it defaults to 0. This is not completely equivalent to slicing the string; the `'^'` pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter endpos limits how far the string will be searched; it will be as if the string is endpos characters long, so only the characters from pos to endpos - 1 will be searched for a match. If endpos is less than pos, no match will be found, otherwise, if rx is a compiled regular expression object, rx.search(string, 0, 50) is equivalent to rx.search(string[:50], 0).

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<_sre.SRE_Match object at ...>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

match(string[, pos[, endpos]])
If zero or more characters at the beginning of string match this regular expression, return a corresponding MatchObject instance. Return None if the string does not match the pattern; note that this is different from a zero-length match.

The optional pos and endpos parameters have the same meaning as for the search() method.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<_sre.SRE_Match object at ...>
```

If you want to locate a match anywhere in string, use search() instead (see also search() vs. match()).

split(string, maxsplit=0)
: Identical to the split() function, using the compiled pattern.

findall(string[, pos[, endpos]])
: Similar to the findall() function, using the compiled pattern, but also accepts optional pos and endpos parameters that limit the search region like for match().

finditer(string[, pos[, endpos]])
: Similar to the finditer() function, using the compiled pattern, but also accepts optional pos and endpos parameters that limit the search region like for match().

sub(repl, string, count=0)
: Identical to the sub() function, using the compiled pattern.

subn(repl, string, count=0)
: Identical to the subn() function, using the compiled pattern.

flags
: The regex matching flags. This is a combination of the flags given to compile() and any (?...) inline flags in the pattern.

groups
: The number of capturing groups in the pattern.

groupindex
: A dictionary mapping any symbolic group names defined by (?P<id>) to group numbers. The dictionary is empty if no symbolic groups were used in the pattern.

pattern
: The pattern string from which the RE object was compiled.

7.2.4 Match Objects

class re.MatchObject
: Match objects always have a boolean value of True. Since match() and search() return None when there is no match, you can test whether there was a match with a simple if statement:

```
match = re.search(pattern, string)
if match:
    process(match)
```

Match objects support the following methods and attributes:

expand(template)
: Return the string obtained by doing backslash substitution on the template string template, as done by the sub() method. Escapes such as \n are converted to the appropriate characters, and numeric backreferences (\1, \2) and named backreferences (\g<1>, \g<name>) are replaced by the contents of the corresponding group.

group([group1, ...])

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, group1 defaults to zero (the whole match is returned). If a groupN argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an IndexError exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is None. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

If the regular expression uses the (?P<name>...) syntax, the groupN arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an IndexError exception is raised.

A moderately complicated example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Named groups can also be referred to by their index:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

If a group matches multiple times, only the last match is accessible:

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

groups([default])

Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The default argument is used for groups that did not participate in the match; it defaults to None. (Incompatibility note: in the original Python 1.5 release, if the tuple was one element long, a string would be returned instead. In later versions (from 1.5.1 on), a singleton tuple is returned in such cases.)

For example:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

If we make the decimal place and everything after it optional, not all groups might participate in the match. These groups will default to None unless the default argument is given:

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

groupdict([default])

Return a dictionary containing all the named subgroups of the match, keyed by the subgroup name. The default argument is used for groups that did not participate in the match; it defaults to None. For example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

start([group])

end([group])

Return the indices of the start and end of the substring matched by group; group defaults to zero (meaning the whole matched substring). Return -1 if group exists but did not contribute to the match. For a match object m, and a group g that did contribute to the match, the substring matched by group g (equivalent to m.group(g)) is

```
m.string[m.start(g):m.end(g)]
```

Note that m.start(group) will equal m.end(group) if group matched a null string. For example, after m = re.search('b(c?)', 'cba'), m.start(0) is 1, m.end(0) is 2, m.start(1) and m.end(1) are both 2, and m.start(2) raises an IndexError exception.

An example that will remove remove_this from email addresses:

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

span([group])

For MatchObject m, return the 2-tuple (m.start(group), m.end(group)). Note that if group did not contribute to the match, this is (-1, -1). group defaults to zero, the entire match.

pos

The value of pos which was passed to the search() or match() method of the RegexObject. This is the index into the string at which the RE engine started looking for a match.

endpos

The value of endpos which was passed to the search() or match() method of the RegexObject. This is the index into the string beyond which the RE engine will not go.

lastindex

The integer index of the last matched capturing group, or None if no group was matched at all. For example, the expressions (a)b, ((a)(b)), and ((ab)) will have lastindex == 1 if applied to the string 'ab', while the expression (a)(b) will have lastindex == 2, if applied to the same string.

lastgroup

The name of the last matched capturing group, or None if the group didn't have a name, or if no group was matched at all.

re
: The regular expression object whose match() or search() method produced this MatchObject instance.

string
: The string passed to match() or search().

7.2.5 Examples

Checking For a Pair

In this example, we'll use the following helper function to display match objects a little more gracefully:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

Suppose you are writing a poker program where a player's hand is represented as a 5-character string with each character representing a card, "a" for ace, "k" for king, "q" for queen, "j" for jack, "t" for 10, and "2" through "9" representing the card with that value.

To see if a given string is a valid hand, one could do the following:

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

That last hand, "727ak", contained a pair, or two of the same valued cards. To match this with a regular expression, one could use backreferences as such:

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # No pairs.
>>> displaymatch(pair.match("354aa"))     # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

To find out what card the pair consists of, one could use the group() method of MatchObject in the following manner:

```
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

Simulating scanf()

Python does not currently have an equivalent to scanf(). Regular expressions are generally more powerful, though also more verbose, than scanf() format strings. The table below offers some more-or-less equivalent mappings between scanf() format tokens and regular expressions.

scanf() Token	Regular Expression
%c	.
%5c	.{5}
%d	[-+]?\d+
%e, %E, %f, %g	[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?
%i	[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)
%o	[-+]?[0-7]+
%s	\S+
%u	\d+
%x, %X	[-+]?(0[xX])?[\dA-Fa-f]+

To extract the filename and numbers from a string like

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

you would use a scanf() format like

```
%s - %d errors, %d warnings
```

The equivalent regular expression would be

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python offers two different primitive operations based on regular expressions: re.match() checks for a match only at the beginning of the string, while re.search() checks for a match anywhere in the string (this is what Perl does by default).

For example:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<_sre.SRE_Match object at ...>
```

Regular expressions beginning with `'^'` can be used with search() to restrict the match at the beginning of the string:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("^c", "abcdef")  # No match
>>> re.search("^a", "abcdef")  # Match
<_sre.SRE_Match object at ...>
```

Note however that in MULTILINE mode match() only matches at the beginning of the string, whereas using search() with a regular expression beginning with `'^'` will match at the beginning of each line.

```
>>> re.match('X', 'A\nB\nX', re.MULTILINE)  # No match
>>> re.search('^X', 'A\nB\nX', re.MULTILINE)  # Match
<_sre.SRE_Match object at ...>
```

Making a Phonebook

split() splits a string into a list delimited by the passed pattern. The method is invaluable for converting textual data into data structures that can be easily read and modified by Python as demonstrated in the following example that creates a phonebook.

First, here is the input. Normally it may come from a file, here we are using triple-quoted string syntax:

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

The entries are separated by one or more newlines. Now we convert the string into a list with each nonempty line having its own entry:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

Finally, split each entry into a list with first name, last name, telephone number, and address. We use the maxsplit parameter of split() because the address has spaces, our splitting pattern, in it:

```
>>> [re.split(":? ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

The :? pattern matches the colon after the last name, so that it does not occur in the result list. With a maxsplit of 4, we could separate the house number from the street name:

```
>>> [re.split(":? ", entry, 4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

Text Munging

sub() replaces every occurrence of a pattern with a string or the result of a function. This example demonstrates using sub() with a function to "munge" text, or randomize the order of all the characters in each word of a sentence except for the first and last characters:

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
```

(continues on next page)

(continued from previous page)

```
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

Finding all Adverbs

findall() matches all occurrences of a pattern, not just the first one as search() does. For example, if a writer wanted to find all of the adverbs in some text, they might use findall() in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

Finding all Adverbs and their Positions

If one wants more information about all matches of a pattern than the matched text, finditer() is useful as it provides instances of MatchObject instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs and their positions in some text, they would use finditer() in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print '%02d-%02d: %s' % (m.start(), m.end(), m.group(0))
07-16: carefully
40-47: quickly
```

Raw String Notation

Raw string notation (r"text") keeps regular expressions sane. Without it, every backslash (`'\'`) in a regular expression would have to be prefixed with another one to escape it. For example, the two following lines of code are functionally identical:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<_sre.SRE_Match object at ...>
>>> re.match("\\W(.)\\1\\W", " ff ")
<_sre.SRE_Match object at ...>
```

When one wants to match a literal backslash, it must be escaped in the regular expression. With raw string notation, this means r"\\". Without raw string notation, one must use "\\\\", making the following lines of code functionally identical:

```
>>> re.match(r"\\", r"\\")
<_sre.SRE_Match object at ...>
>>> re.match("\\\\", r"\\")
<_sre.SRE_Match object at ...>
```

7.3 struct — Interpret strings as packed binary data

This module performs conversions between Python values and C structs represented as Python strings. This can be used in handling binary data stored in files or from network connections, among other sources. It uses

Format Strings as compact descriptions of the layout of the C structs and the intended conversion to/from Python values.

Note: By default, the result of packing a given C struct includes pad bytes in order to maintain proper alignment for the C types involved; similarly, alignment is taken into account when unpacking. This behavior is chosen so that the bytes of a packed struct correspond exactly to the layout in memory of the corresponding C struct. To handle platform-independent data formats or omit implicit pad bytes, use standard size and alignment instead of native size and alignment: see Byte Order, Size, and Alignment for details.

7.3.1 Functions and Exceptions

The module defines the following exception and functions:

exception struct.error
: Exception raised on various occasions; argument is a string describing what is wrong.

struct.pack(fmt, v1, v2, ...)
: Return a string containing the values v1, v2, ... packed according to the given format. The arguments must match the values required by the format exactly.

struct.pack_into(fmt, buffer, offset, v1, v2, ...)
: Pack the values v1, v2, ... according to the given format, write the packed bytes into the writable buffer starting at offset. Note that the offset is a required argument.

  New in version 2.5.

struct.unpack(fmt, string)
: Unpack the string (presumably packed by pack(fmt, ...)) according to the given format. The result is a tuple even if it contains exactly one item. The string must contain exactly the amount of data required by the format (len(string) must equal calcsize(fmt)).

struct.unpack_from(fmt, buffer[, offset=0])
: Unpack the buffer according to the given format. The result is a tuple even if it contains exactly one item. The buffer must contain at least the amount of data required by the format (len(buffer[offset:]) must be at least calcsize(fmt)).

  New in version 2.5.

struct.calcsize(fmt)
: Return the size of the struct (and hence of the string) corresponding to the given format.

7.3.2 Format Strings

Format strings are the mechanism used to specify the expected layout when packing and unpacking data. They are built up from Format Characters, which specify the type of data being packed/unpacked. In addition, there are special characters for controlling the Byte Order, Size, and Alignment.

Byte Order, Size, and Alignment

By default, C types are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler).

Alternatively, the first character of the format string can be used to indicate the byte order, size and alignment of the packed data, according to the following table:

Character	Byte order	Size	Alignment
@	native	native	native
=	native	standard	none
<	little-endian	standard	none
>	big-endian	standard	none
!	network (= big-endian)	standard	none

If the first character is not one of these, `'@'` is assumed.

Native byte order is big-endian or little-endian, depending on the host system. For example, Intel x86 and AMD64 (x86-64) are little-endian; Motorola 68000 and PowerPC G5 are big-endian; ARM and Intel Itanium feature switchable endianness (bi-endian). Use sys.byteorder to check the endianness of your system.

Native size and alignment are determined using the C compiler's sizeof expression. This is always combined with native byte order.

Standard size depends only on the format character; see the table in the Format Characters section.

Note the difference between `'@'` and `'='`: both use native byte order, but the size and alignment of the latter is standardized.

The form `'!'` is available for those poor souls who claim they can't remember whether network byte order is big-endian or little-endian.

There is no way to indicate non-native byte order (force byte-swapping); use the appropriate choice of `'<'` or `'>'`.

Notes:

(1) Padding is only automatically added between successive structure members. No padding is added at the beginning or the end of the encoded struct.

(2) No padding is added when using non-native size and alignment, e.g. with '<', '>', '=', and '!'.

(3) To align the end of a structure to the alignment requirement of a particular type, end the format with the code for that type with a repeat count of zero. See Examples.

Format Characters

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types. The 'Standard size' column refers to the size of the packed value in bytes when using standard size; that is, when the format string starts with one of `'<'`, `'>'`, `'!'` or `'='`. When using native size, the size of the packed value is platform-dependent.

Format	C Type	Python type	Standard size	Notes
x	pad byte	no value		
c	char	string of length 1	1	
b	signed char	integer	1	(3)
B	unsigned char	integer	1	(3)
?	_Bool	bool	1	(1)
h	short	integer	2	(3)
H	unsigned short	integer	2	(3)
i	int	integer	4	(3)
I	unsigned int	integer	4	(3)
l	long	integer	4	(3)
L	unsigned long	integer	4	(3)
q	long long	integer	8	(2), (3)
Q	unsigned long long	integer	8	(2), (3)
f	float	float	4	(4)
d	double	float	8	(4)
s	char[]	string		
p	char[]	string		
P	void *	integer		(5), (3)

Notes:

(1) The `'?'` conversion code corresponds to the _Bool type defined by C99. If this type is not available, it is simulated using a char. In standard mode, it is always represented by one byte.

New in version 2.6.

(2) The `'q'` and `'Q'` conversion codes are available in native mode only if the platform C compiler supports C long long, or, on Windows, __int64. They are always available in standard modes.

New in version 2.2.

(3) When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a __index__() method then that method is called to convert the argument to an integer before packing. If no __index__() method exists, or the call to __index__() raises TypeError, then the __int__() method is tried. However, the use of __int__() is deprecated, and will raise DeprecationWarning.

Changed in version 2.7: Use of the __index__() method for non-integers is new in 2.7.

Changed in version 2.7: Prior to version 2.7, not all integer conversion codes would use the __int__() method to convert, and DeprecationWarning was raised only for float arguments.

(4) For the `'f'` and `'d'` conversion codes, the packed representation uses the IEEE 754 binary32 (for `'f'`) or binary64 (for `'d'`) format, regardless of the floating-point format used by the platform.

(5) The `'P'` format character is only available for the native byte ordering (selected as the default or with the `'@'` byte order character). The byte order character `'='` chooses to use little- or big-endian ordering based on the host system. The struct module does not interpret this as native ordering, so the `'P'` format is not available.

A format character may be preceded by an integral repeat count. For example, the format string `'4h'` means exactly the same as `'hhhh'`.

Whitespace characters between formats are ignored; a count and its format must not contain whitespace though.

For the 's' format character, the count is interpreted as the size of the string, not a repeat count like for the other format characters; for example, '10s' means a single 10-byte string, while '10c' means 10 characters. If a count is not given, it defaults to 1. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting string always has exactly the specified number of bytes. As a special case, '0s' means a single, empty string (while '0c' means 0 characters).

The 'p' format character encodes a "Pascal string", meaning a short variable-length string stored in a fixed number of bytes, given by the count. The first byte stored is the length of the string, or 255, whichever is smaller. The bytes of the string follow. If the string passed in to pack() is too long (longer than the count minus 1), only the leading count-1 bytes of the string are stored. If the string is shorter than count-1, it is padded with null bytes so that exactly count bytes in all are used. Note that for unpack(), the 'p' format character consumes count bytes, but that the string returned can never contain more than 255 characters.

For the 'P' format character, the return value is a Python integer or long integer, depending on the size needed to hold a pointer when it has been cast to an integer type. A NULL pointer will always be returned as the Python integer 0. When packing pointer-sized values, Python integer or long integer objects may be used. For example, the Alpha and Merced processors use 64-bit pointer values, meaning a Python long integer will be used to hold the pointer; other platforms use 32-bit pointers and will use a Python integer.

For the '?' format character, the return value is either True or False. When packing, the truth value of the argument object is used. Either 0 or 1 in the native or standard bool representation will be packed, and any non-zero value will be True when unpacking.

Examples

Note: All examples assume a native byte order, size, and alignment with a big-endian machine.

A basic example of packing/unpacking three integers:

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', '\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

Unpacked fields can be named by assigning them to variables or by wrapping the result in a named tuple:

```
>>> record = 'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name='raymond   ', serialnum=4658, school=264, gradelevel=8)
```

The ordering of format characters may have an impact on size since the padding needed to satisfy alignment requirements is different:

```
>>> pack('ci', '*', 0x12131415)
'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, '*')
'\x12\x13\x14\x15*'
>>> calcsize('ci')
```

(continues on next page)

(continued from previous page)

```
8
>>> calcsize('ic')
5
```

The following format `'llh0l'` specifies two pad bytes at the end, assuming longs are aligned on 4-byte boundaries:

```
>>> pack('llh0l', 1, 2, 3)
'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

This only works when native size and alignment are in effect; standard size and alignment does not enforce any alignment.

See also:

Module array Packed binary storage of homogeneous data.

Module xdrlib Packing and unpacking of XDR data.

7.3.3 Classes

The struct module also defines the following type:

class struct.Struct(format)
: Return a new Struct object which writes and reads binary data according to the format string format. Creating a Struct object once and calling its methods is more efficient than calling the struct functions with the same format since the format string only needs to be compiled once.

  New in version 2.5.

  Compiled Struct objects support the following methods and attributes:

  pack(v1, v2, ...)
  : Identical to the pack() function, using the compiled format. (len(result) will equal self.size.)

  pack_into(buffer, offset, v1, v2, ...)
  : Identical to the pack_into() function, using the compiled format.

  unpack(string)
  : Identical to the unpack() function, using the compiled format. (len(string) must equal self.size).

  unpack_from(buffer, offset=0)
  : Identical to the unpack_from() function, using the compiled format. (len(buffer[offset:]) must be at least self.size).

  format
  : The format string used to construct this Struct object.

  size
  : The calculated size of the struct (and hence of the string) corresponding to format.

7.4 difflib — Helpers for computing deltas

New in version 2.1.

This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce difference information in various formats, including HTML and context and unified diffs. For comparing directories and files, see also, the filecmp module.

class difflib.SequenceMatcher
This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are hashable. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980's by Ratcliff and Obershelp under the hyperbolic name "gestalt pattern matching." The idea is to find the longest contiguous matching subsequence that contains no "junk" elements (the Ratcliff and Obershelp algorithm doesn't address junk). The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that "look right" to people.

Timing: The basic Ratcliff-Obershelp algorithm is cubic time in the worst case and quadratic time in the expected case. SequenceMatcher is quadratic time for the worst case and has expected-case behavior dependent in a complicated way on how many elements the sequences have in common; best case time is linear.

Automatic junk heuristic: SequenceMatcher supports a heuristic that automatically treats certain sequence items as junk. The heuristic counts how many times each individual item appears in the sequence. If an item's duplicates (after the first one) account for more than 1% of the sequence and the sequence is at least 200 items long, this item is marked as "popular" and is treated as junk for the purpose of sequence matching. This heuristic can be turned off by setting the autojunk argument to False when creating the SequenceMatcher.

New in version 2.7.1: The autojunk parameter.

class difflib.Differ
This is a class for comparing sequences of lines of text, and producing human-readable differences or deltas. Differ uses SequenceMatcher both to compare sequences of lines, and to compare sequences of characters within similar (near-matching) lines.

Each line of a Differ delta begins with a two-letter code:

Code	Meaning
'- '	line unique to sequence 1
'+ '	line unique to sequence 2
' '	line common to both sequences
'? '	line not present in either input sequence

Lines beginning with '?' attempt to guide the eye to intraline differences, and were not present in either input sequence. These lines can be confusing if the sequences contain tab characters.

class difflib.HtmlDiff
This class can be used to create an HTML table (or a complete HTML file containing the table) showing a side by side, line by line comparison of text with inter-line and intra-line change highlights. The table can be generated in either full or contextual difference mode.

The constructor for this class is:

__init__(tabsize=8, wrapcolumn=None, linejunk=None, charjunk=IS_CHARACTER_JUNK)
Initializes instance of HtmlDiff.

tabsize is an optional keyword argument to specify tab stop spacing and defaults to 8.

wrapcolumn is an optional keyword to specify column number where lines are broken and wrapped, defaults to None where lines are not wrapped.

linejunk and charjunk are optional keyword arguments passed into ndiff() (used by HtmlDiff to generate the side by side HTML differences). See ndiff() documentation for argument default values and descriptions.

The following methods are public:

make_file(fromlines, tolines [, fromdesc][, todesc][, context][, numlines])
Compares fromlines and tolines (lists of strings) and returns a string which is a complete HTML file containing a table showing line by line differences with inter-line and intra-line changes highlighted.

fromdesc and todesc are optional keyword arguments to specify from/to file column header strings (both default to an empty string).

context and numlines are both optional keyword arguments. Set context to True when contextual differences are to be shown, else the default is False to show the full files. numlines defaults to 5. When context is True numlines controls the number of context lines which surround the difference highlights. When context is False numlines controls the number of lines which are shown before a difference highlight when using the "next" hyperlinks (setting to zero would cause the "next" hyperlinks to place the next difference highlight at the top of the browser without any leading context).

make_table(fromlines, tolines [, fromdesc][, todesc][, context][, numlines])
Compares fromlines and tolines (lists of strings) and returns a string which is a complete HTML table showing line by line differences with inter-line and intra-line changes highlighted.

The arguments for this method are the same as those for the make_file() method.

Tools/scripts/diff.py is a command-line front-end to this class and contains a good example of its use.

New in version 2.4.

difflib.context_diff(a, b[, fromfile][, tofile][, fromfiledate][, tofiledate][, n][, lineterm])
Compare a and b (lists of strings); return a delta (a generator generating the delta lines) in context diff format.

Context diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a before/after style. The number of context lines is set by n which defaults to three.

By default, the diff control lines (those with *** or ---) are created with a trailing newline. This is helpful so that inputs created from file.readlines() result in diffs that are suitable for use with file.writelines() since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the lineterm argument to "" so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for fromfile, tofile, fromfiledate, and tofiledate. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> for line in context_diff(s1, s2, fromfile='before.py', tofile='after.py'):
...     sys.stdout.write(line)  # doctest: +NORMALIZE_WHITESPACE
*** before.py
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
```

(continues on next page)

(continued from previous page)

```
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

See A command-line interface to difflib for a more detailed example.

New in version 2.3.

difflib.get_close_matches(word, possibilities[, n][, cutoff])
Return a list of the best "good enough" matches. word is a sequence for which close matches are desired (typically a string), and possibilities is a list of sequences against which to match word (typically a list of strings).

Optional argument n (default 3) is the maximum number of close matches to return; n must be greater than 0.

Optional argument cutoff (default 0.6) is a float in the range [0, 1]. Possibilities that don't score at least that similar to word are ignored.

The best (no more than n) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('apple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

difflib.ndiff(a, b[, linejunk][, charjunk])
Compare a and b (lists of strings); return a Differ-style delta (a generator generating the delta lines).

Optional keyword parameters linejunk and charjunk are for filter functions (or None):

linejunk: A function that accepts a single string argument, and returns true if the string is junk, or false if not. The default is (None), starting with Python 2.3. Before then, the default was the module-level function IS_LINE_JUNK(), which filters out lines without visible characters, except for at most one pound character ('#'). As of Python 2.3, the underlying SequenceMatcher class does a dynamic analysis of which lines are so frequent as to constitute noise, and this usually works better than the pre-2.3 default.

charjunk: A function that accepts a character (a string of length 1), and returns if the character is junk, or false if not. The default is module-level function IS_CHARACTER_JUNK(), which filters out whitespace characters (a blank or tab; note: bad idea to include newline in this!).

Tools/scripts/ndiff.py is a command-line front-end to this function.

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
...              'ore\ntree\nemu\n'.splitlines(1))
>>> print ''.join(diff),
- one
?  ^
```

(continues on next page)

(continued from previous page)

```
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

difflib.restore(sequence, which)

Return one of the two sequences that generated a delta.

Given a sequence produced by Differ.compare() or ndiff(), extract lines originating from file 1 or 2 (parameter which), stripping off line prefixes.

Example:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
...              'ore\ntree\nemu\n'.splitlines(1))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print ''.join(restore(diff, 1)),
one
two
three
>>> print ''.join(restore(diff, 2)),
ore
tree
emu
```

difflib.unified_diff(a, b[, fromfile][, tofile][, fromfiledate][, tofiledate][, n][, lineterm])

Compare a and b (lists of strings); return a delta (a generator generating the delta lines) in unified diff format.

Unified diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in an inline style (instead of separate before/after blocks). The number of context lines is set by n which defaults to three.

By default, the diff control lines (those with ---, +++, or @@) are created with a trailing newline. This is helpful so that inputs created from file.readlines() result in diffs that are suitable for use with file.writelines() since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the lineterm argument to "" so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for fromfile, tofile, fromfiledate, and tofiledate. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> for line in unified_diff(s1, s2, fromfile='before.py', tofile='after.py'):
...     sys.stdout.write(line)   # doctest: +NORMALIZE_WHITESPACE
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
```

(continues on next page)

(continued from previous page)

```
+eggy
+hamster
 guido
```

See A command-line interface to difflib for a more detailed example.

New in version 2.3.

difflib.IS_LINE_JUNK(line)
Return true for ignorable lines. The line line is ignorable if line is blank or contains a single '#', otherwise it is not ignorable. Used as a default for parameter linejunk in ndiff() before Python 2.3.

difflib.IS_CHARACTER_JUNK(ch)
Return true for ignorable characters. The character ch is ignorable if ch is a space or tab, otherwise it is not ignorable. Used as a default for parameter charjunk in ndiff().

See also:

Pattern Matching: The Gestalt Approach Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in Dr. Dobb's Journal in July, 1988.

7.4.1 SequenceMatcher Objects

The SequenceMatcher class has this constructor:

class difflib.SequenceMatcher(isjunk=None, a='', b='', autojunk=True)
Optional argument isjunk must be None (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is "junk" and should be ignored. Passing None for isjunk is equivalent to passing lambda x: 0; in other words, no elements are ignored. For example, pass:

```
lambda x: x in " \t"
```

if you're comparing lines as sequences of characters, and don't want to synch up on blanks or hard tabs.

The optional arguments a and b are sequences to be compared; both default to empty strings. The elements of both sequences must be hashable.

The optional argument autojunk can be used to disable the automatic junk heuristic.

New in version 2.7.1: The autojunk parameter.

SequenceMatcher objects have the following methods:

set_seqs(a, b)
Set the two sequences to be compared.

SequenceMatcher computes and caches detailed information about the second sequence, so if you want to compare one sequence against many sequences, use set_seq2() to set the commonly used sequence once and call set_seq1() repeatedly, once for each of the other sequences.

set_seq1(a)
Set the first sequence to be compared. The second sequence to be compared is not changed.

set_seq2(b)
Set the second sequence to be compared. The first sequence to be compared is not changed.

find_longest_match(alo, ahi, blo, bhi)
Find longest matching block in a[alo:ahi] and b[blo:bhi].

If isjunk was omitted or None, find_longest_match() returns (i, j, k) such that a[i:i+k] is equal to b[j:j+k], where alo <= i <= i+k <= ahi and blo <= j <= j+k <= bhi. For all (i', j', k') meeting those conditions, the additional conditions k >= k', i <= i', and if i == i', j <= j' are also met. In other words, of all maximal matching blocks, return one that starts earliest in a, and of all those maximal matching blocks that start earliest in a, return the one that starts earliest in b.

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

If isjunk was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.

Here's the same example as before, but considering blanks to be junk. That prevents ' abcd' from matching the ' abcd' at the tail end of the second sequence directly. Instead only the 'abcd' can match, and matches the leftmost 'abcd' in the second sequence:

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

If no blocks match, this returns (alo, blo, 0).

Changed in version 2.6: This method returns a named tuple Match(a, b, size).

get_matching_blocks()
Return list of triples describing non-overlapping matching subsequences. Each triple is of the form (i, j, n), and means that a[i:i+n] == b[j:j+n]. The triples are monotonically increasing in i and j.

The last triple is a dummy, and has the value (len(a), len(b), 0). It is the only triple with n == 0. If (i, j, n) and (i', j', n') are adjacent triples in the list, and the second is not the last triple in the list, then i+n < i' or j+n < j'; in other words, adjacent triples always describe non-adjacent equal blocks.

Changed in version 2.5: The guarantee that adjacent triples always describe non-adjacent blocks was implemented.

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()
Return list of 5-tuples describing how to turn a into b. Each tuple is of the form (tag, i1, i2, j1, j2). The first tuple has i1 == j1 == 0, and remaining tuples have i1 equal to the i2 from the preceding tuple, and, likewise, j1 equal to the previous j2.

The tag values are strings, with these meanings:

Value	Meaning
'replace'	a[i1:i2] should be replaced by b[j1:j2].
'delete'	a[i1:i2] should be deleted. Note that j1 == j2 in this case.
'insert'	b[j1:j2] should be inserted at a[i1:i1]. Note that i1 == i2 in this case.
'equal'	a[i1:i2] == b[j1:j2] (the sub-sequences are equal).

For example:

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...    print ("%7s a[%d:%d] (%s) b[%d:%d] (%s)" %
...           (tag, i1, i2, a[i1:i2], j1, j2, b[j1:j2]))
 delete a[0:1] (q) b[0:0] ()
  equal a[1:3] (ab) b[0:2] (ab)
replace a[3:4] (x) b[2:3] (y)
  equal a[4:6] (cd) b[3:5] (cd)
 insert a[6:6] () b[5:6] (f)
```

get_grouped_opcodes([n])
: Return a generator of groups with up to n lines of context.

  Starting with the groups returned by get_opcodes(), this method splits out smaller change clusters and eliminates intervening ranges which have no changes.

  The groups are returned in the same format as get_opcodes().

  New in version 2.3.

ratio()
: Return a measure of the sequences' similarity as a float in the range [0, 1].

  Where T is the total number of elements in both sequences, and M is the number of matches, this is 2.0*M / T. Note that this is 1.0 if the sequences are identical, and 0.0 if they have nothing in common.

  This is expensive to compute if get_matching_blocks() or get_opcodes() hasn't already been called, in which case you may want to try quick_ratio() or real_quick_ratio() first to get an upper bound.

quick_ratio()
: Return an upper bound on ratio() relatively quickly.

real_quick_ratio()
: Return an upper bound on ratio() very quickly.

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although quick_ratio() and real_quick_ratio() are always at least as large as ratio():

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

7.4.2 SequenceMatcher Examples

This example compares two strings, considering blanks to be "junk:"

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

ratio() returns a float in [0, 1], measuring the similarity of the sequences. As a rule of thumb, a ratio() value over 0.6 means the sequences are close matches:

```
>>> print round(s.ratio(), 3)
0.866
```

If you're only interested in where the sequences match, get_matching_blocks() is handy:

```
>>> for block in s.get_matching_blocks():
...     print "a[%d] and b[%d] match for %d elements" % block
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

Note that the last tuple returned by get_matching_blocks() is always a dummy, (len(a), len(b), 0), and this is the only case in which the last tuple element (number of elements matched) is 0.

If you want to know how to change the first sequence into the second, use get_opcodes():

```
>>> for opcode in s.get_opcodes():
...     print "%6s a[%d:%d] b[%d:%d]" % opcode
 equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

See also:

- The get_close_matches() function in this module which shows how simple code building on SequenceMatcher can be used to do useful work.
- Simple version control recipe for a small application built with SequenceMatcher.

7.4.3 Differ Objects

Note that Differ-generated deltas make no claim to be minimal diffs. To the contrary, minimal diffs are often counter-intuitive, because they synch up anywhere possible, sometimes accidental matches 100 pages apart. Restricting synch points to contiguous matches preserves some notion of locality, at the occasional cost of producing a longer diff.

The Differ class has this constructor:

class difflib.Differ([linejunk[, charjunk]])

Optional keyword parameters linejunk and charjunk are for filter functions (or None):

linejunk: A function that accepts a single string argument, and returns true if the string is junk. The default is None, meaning that no line is considered junk.

charjunk: A function that accepts a single character argument (a string of length 1), and returns true if the character is junk. The default is None, meaning that no character is considered junk.

Differ objects are used (deltas generated) via a single method:

compare(a, b)

Compare two sequences of lines, and generate the delta (a sequence of lines).

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the readlines() method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the writelines() method of a file-like object.

7.4.4 Differ Example

This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the readlines() method of file-like objects):

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(1)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(1)
```

Next we instantiate a Differ object:

```
>>> d = Differ()
```

Note that when instantiating a Differ object we may pass functions to filter out line and character "junk." See the Differ() constructor for details.

Finally, we compare the two:

```
>>> result = list(d.compare(text1, text2))
```

result is a list of strings, so let's pretty-print it:

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3.   Simple is better than complex.\n',
 '?     ++\n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^\n',
 '+   4. Complicated is better than complex.\n',
 '?           +++ ^                         ^\n',
 '+   5. Flat is better than nested.\n']
```

As a single multi-line string it looks like this:

```
>>> import sys
>>> sys.stdout.writelines(result)
    1. Beautiful is better than ugly.
```

(continues on next page)

(continued from previous page)

```
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+    3.   Simple is better than complex.
?     ++
-   4. Complex is better than complicated.
?            ^                   ---- ^
+    4. Complicated is better than complex.
?          ++++ ^                     ^
+    5. Flat is better than nested.
```

7.4.5 A command-line interface to difflib

This example shows how to use difflib to create a diff-like utility. It is also contained in the Python source distribution, as Tools/scripts/diff.py.

```
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

import sys, os, time, difflib, optparse

def main():
    # Configure the option parser
    usage = "usage: %prog [options] fromfile tofile"
    parser = optparse.OptionParser(usage)
    parser.add_option("-c", action="store_true", default=False,
                      help='Produce a context format diff (default)')
    parser.add_option("-u", action="store_true", default=False,
                      help='Produce a unified format diff')
    hlp = 'Produce HTML side by side diff (can use -c and -l in conjunction)'
    parser.add_option("-m", action="store_true", default=False, help=hlp)
    parser.add_option("-n", action="store_true", default=False,
                      help='Produce a ndiff format diff')
    parser.add_option("-l", "--lines", type="int", default=3,
                      help='Set number of context lines (default 3)')
    (options, args) = parser.parse_args()

    if len(args) == 0:
        parser.print_help()
        sys.exit(1)
    if len(args) != 2:
        parser.error("need to specify both a fromfile and tofile")

    n = options.lines
    fromfile, tofile = args # as specified in the usage string

    # we're passing these as arguments to the diff function
    fromdate = time.ctime(os.stat(fromfile).st_mtime)
    todate = time.ctime(os.stat(tofile).st_mtime)
    with open(fromfile, 'U') as f:
```

(continues on next page)

(continued from previous page)

```
        fromlines = f.readlines()
    with open(tofile, 'U') as f:
        tolines = f.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile,
                                    fromdate, todate, n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
        diff = difflib.HtmlDiff().make_file(fromlines, tolines, fromfile,
                                            tofile, context=options.c,
                                            numlines=n)
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile,
                                    fromdate, todate, n=n)

    # we're using writelines because diff is a generator
    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

7.5 StringIO — Read and write strings as files

This module implements a file-like class, StringIO, that reads and writes a string buffer (also known as memory files). See the description of file objects for operations (section File Objects). (For standard strings, see str and unicode.)

class StringIO.StringIO([buffer])
: When a StringIO object is created, it can be initialized to an existing string by passing the string to the constructor. If no string is given, the StringIO will start empty. In both cases, the initial file position starts at zero.

  The StringIO object can accept either Unicode or 8-bit strings, but mixing the two may take some care. If both are used, 8-bit strings that cannot be interpreted as 7-bit ASCII (that use the 8th bit) will cause a UnicodeError to be raised when getvalue() is called.

The following methods of StringIO objects require special mention:

StringIO.getvalue()
: Retrieve the entire contents of the "file" at any time before the StringIO object's close() method is called. See the note above for information about mixing Unicode and 8-bit strings; such mixing can cause this method to raise UnicodeError.

StringIO.close()
: Free the memory buffer. Attempting to do further operations with a closed StringIO object will raise a ValueError.

Example usage:

```
import StringIO

output = StringIO.StringIO()
output.write('First line.\n')
```

(continues on next page)

(continued from previous page)

```
print >>output, 'Second line.'

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

7.6 cStringIO — Faster version of StringIO

The module cStringIO provides an interface similar to that of the StringIO module. Heavy use of StringIO.StringIO objects can be made more efficient by using the function StringIO() from this module instead.

cStringIO.StringIO([s])
: Return a StringIO-like stream for reading or writing.

  Since this is a factory function which returns objects of built-in types, there's no way to build your own version using subclassing. It's not possible to set attributes on it. Use the original StringIO module in those cases.

  Unlike the StringIO module, this module is not able to accept Unicode strings that cannot be encoded as plain ASCII strings.

  Another difference from the StringIO module is that calling StringIO() with a string parameter creates a read-only object. Unlike an object created without a string parameter, it does not have write methods. These objects are not generally visible. They turn up in tracebacks as StringI and StringO.

The following data objects are provided as well:

cStringIO.InputType
: The type object of the objects created by calling StringIO() with a string parameter.

cStringIO.OutputType
: The type object of the objects returned by calling StringIO() with no parameters.

There is a C API to the module as well; refer to the module source for more information.

Example usage:

```
import cStringIO

output = cStringIO.StringIO()
output.write('First line.\n')
print >>output, 'Second line.'

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

7.7 textwrap — Text wrapping and filling

New in version 2.3.

Source code: Lib/textwrap.py

The textwrap module provides two convenience functions, wrap() and fill(), as well as TextWrapper, the class that does all the work, and a utility function dedent(). If you're just wrapping or filling one or two text strings, the convenience functions should be good enough; otherwise, you should use an instance of TextWrapper for efficiency.

textwrap.wrap(text[, width[, ...]])
: Wraps the single paragraph in text (a string) so every line is at most width characters long. Returns a list of output lines, without final newlines.

  Optional keyword arguments correspond to the instance attributes of TextWrapper, documented below. width defaults to 70.

  See the TextWrapper.wrap() method for additional details on how wrap() behaves.

textwrap.fill(text[, width[, ...]])
: Wraps the single paragraph in text, and returns a single string containing the wrapped paragraph. fill() is shorthand for

```
"\n".join(wrap(text, ...))
```

  In particular, fill() accepts exactly the same keyword arguments as wrap().

Both wrap() and fill() work by creating a TextWrapper instance and calling a single method on it. That instance is not reused, so for applications that wrap/fill many text strings, it will be more efficient for you to create your own TextWrapper object.

Text is preferably wrapped on whitespaces and right after the hyphens in hyphenated words; only then will long words be broken if necessary, unless TextWrapper.break_long_words is set to false.

An additional utility function, dedent(), is provided to remove indentation from strings that have unwanted whitespace to the left of the text.

textwrap.dedent(text)
: Remove any common leading whitespace from every line in text.

  This can be used to make triple-quoted strings line up with the left edge of the display, while still presenting them in the source code in indented form.

  Note that tabs and spaces are both treated as whitespace, but they are not equal: the lines " hello" and "\thello" are considered to have no common leading whitespace. (This behaviour is new in Python 2.5; older versions of this module incorrectly expanded tabs before searching for common leading whitespace.)

  For example:

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print repr(s)          # prints '    hello\n      world\n    '
    print repr(dedent(s))  # prints 'hello\n  world\n'
```

class textwrap.TextWrapper(...)

The TextWrapper constructor accepts a number of optional keyword arguments. Each argument corresponds to one instance attribute, so for example

```
wrapper = TextWrapper(initial_indent="* ")
```

is the same as

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

You can re-use the same TextWrapper object many times, and you can change any of its options through direct assignment to instance attributes between uses.

The TextWrapper instance attributes (and keyword arguments to the constructor) are as follows:

width
(default: 70) The maximum length of wrapped lines. As long as there are no individual words in the input text longer than width, TextWrapper guarantees that no output line will be longer than width characters.

expand_tabs
(default: True) If true, then all tab characters in text will be expanded to spaces using the expandtabs() method of text.

replace_whitespace
(default: True) If true, after tab expansion but before wrapping, the wrap() method will replace each whitespace character with a single space. The whitespace characters replaced are as follows: tab, newline, vertical tab, formfeed, and carriage return (`'\t\n\v\f\r'`).

Note: If expand_tabs is false and replace_whitespace is true, each tab character will be replaced by a single space, which is not the same as tab expansion.

Note: If replace_whitespace is false, newlines may appear in the middle of a line and cause strange output. For this reason, text should be split into paragraphs (using str.splitlines() or similar) which are wrapped separately.

---

drop_whitespace
(default: True) If true, whitespace at the beginning and ending of every line (after wrapping but before indenting) is dropped. Whitespace at the beginning of the paragraph, however, is not dropped if non-whitespace follows it. If whitespace being dropped takes up an entire line, the whole line is dropped.

New in version 2.6: Whitespace was always dropped in earlier versions.

initial_indent
(default: `''`) String that will be prepended to the first line of wrapped output. Counts towards the length of the first line. The empty string is not indented.

subsequent_indent
(default: `''`) String that will be prepended to all lines of wrapped output except the first. Counts towards the length of each line except the first.

fix_sentence_endings
(default: False) If true, TextWrapper attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a

monospaced font. However, the sentence detection algorithm is imperfect: it assumes that a sentence ending consists of a lowercase letter followed by one of `'.'`, `'!'`, or `'?'`, possibly followed by one of `'"'` or `"'"`, followed by a space. One problem with this is algorithm is that it is unable to detect the difference between "Dr." in

```
[...] Dr. Frankenstein's monster [...]
```

and "Spot." in

```
[...] See Spot. See Spot run [...]
```

fix_sentence_endings is false by default.

Since the sentence detection algorithm relies on string.lowercase for the definition of "lowercase letter," and a convention of using two spaces after a period to separate sentences on the same line, it is specific to English-language texts.

break_long_words
: (default: True) If true, then words longer than width will be broken in order to ensure that no lines are longer than width. If it is false, long words will not be broken, and some lines may be longer than width. (Long words will be put on a line by themselves, in order to minimize the amount by which width is exceeded.)

break_on_hyphens
: (default: True) If true, wrapping will occur preferably on whitespaces and right after hyphens in compound words, as it is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks, but you need to set break_long_words to false if you want truly insecable words. Default behaviour in previous versions was to always allow breaking hyphenated words.

  New in version 2.6.

TextWrapper also provides two public methods, analogous to the module-level convenience functions:

wrap(text)
: Wraps the single paragraph in text (a string) so every line is at most width characters long. All wrapping options are taken from instance attributes of the TextWrapper instance. Returns a list of output lines, without final newlines. If the wrapped output has no content, the returned list is empty.

fill(text)
: Wraps the single paragraph in text, and returns a single string containing the wrapped paragraph.

7.8 codecs — Codec registry and base classes

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry which manages the codec and error handling lookup process.

It defines the following functions:

codecs.encode(obj[, encoding[, errors]])
: Encodes obj using the codec registered for encoding. The default encoding is `'ascii'`.

  Errors may be given to set the desired error handling scheme. The default error handler is `'strict'` meaning that encoding errors raise ValueError (or a more codec specific subclass, such as UnicodeEncodeError). Refer to Codec Base Classes for more information on codec error handling.

  New in version 2.4.

codecs.decode(obj[, encoding[, errors]])
Decodes obj using the codec registered for encoding. The default encoding is 'ascii'.

Errors may be given to set the desired error handling scheme. The default error handler is 'strict' meaning that decoding errors raise ValueError (or a more codec specific subclass, such as UnicodeDecodeError). Refer to Codec Base Classes for more information on codec error handling.

New in version 2.4.

codecs.register(search_function)
Register a codec search function. Search functions are expected to take one argument, the encoding name in all lower case letters, and return a CodecInfo object having the following attributes:

- name The name of the encoding;
- encode The stateless encoding function;
- decode The stateless decoding function;
- incrementalencoder An incremental encoder class or factory function;
- incrementaldecoder An incremental decoder class or factory function;
- streamwriter A stream writer class or factory function;
- streamreader A stream reader class or factory function.

The various functions or classes take the following arguments:

encode and decode: These must be functions or methods which have the same interface as the encode()/decode() methods of Codec instances (see Codec Interface). The functions/methods are expected to work in a stateless mode.

incrementalencoder and incrementaldecoder: These have to be factory functions providing the following interface:

factory(errors='strict')

The factory functions must return objects providing the interfaces defined by the base classes IncrementalEncoder and IncrementalDecoder, respectively. Incremental codecs can maintain state.

streamreader and streamwriter: These have to be factory functions providing the following interface:

factory(stream, errors='strict')

The factory functions must return objects providing the interfaces defined by the base classes StreamReader and StreamWriter, respectively. Stream codecs can maintain state.

Possible values for errors are

- 'strict': raise an exception in case of an encoding error
- 'replace': replace malformed data with a suitable replacement marker, such as '?' or '\ufffd'
- 'ignore': ignore malformed data and continue without further notice
- 'xmlcharrefreplace': replace with the appropriate XML character reference (for encoding only)
- 'backslashreplace': replace with backslashed escape sequences (for encoding only)

as well as any other error handling name defined via register_error().

In case a search function cannot find a given encoding, it should return None.

codecs.lookup(encoding)
Looks up the codec info in the Python codec registry and returns a CodecInfo object as defined above.

Encodings are first looked up in the registry's cache. If not found, the list of registered search functions is scanned. If no CodecInfo object is found, a LookupError is raised. Otherwise, the CodecInfo object is stored in the cache and returned to the caller.

To simplify access to the various codecs, the module provides these additional functions which use lookup() for the codec lookup:

codecs.getencoder(encoding)
Look up the codec for the given encoding and return its encoder function.

Raises a LookupError in case the encoding cannot be found.

codecs.getdecoder(encoding)
Look up the codec for the given encoding and return its decoder function.

Raises a LookupError in case the encoding cannot be found.

codecs.getincrementalencoder(encoding)
Look up the codec for the given encoding and return its incremental encoder class or factory function.

Raises a LookupError in case the encoding cannot be found or the codec doesn't support an incremental encoder.

New in version 2.5.

codecs.getincrementaldecoder(encoding)
Look up the codec for the given encoding and return its incremental decoder class or factory function.

Raises a LookupError in case the encoding cannot be found or the codec doesn't support an incremental decoder.

New in version 2.5.

codecs.getreader(encoding)
Look up the codec for the given encoding and return its StreamReader class or factory function.

Raises a LookupError in case the encoding cannot be found.

codecs.getwriter(encoding)
Look up the codec for the given encoding and return its StreamWriter class or factory function.

Raises a LookupError in case the encoding cannot be found.

codecs.register_error(name, error_handler)
Register the error handling function error_handler under the name name. error_handler will be called during encoding and decoding in case of an error, when name is specified as the errors parameter.

For encoding error_handler will be called with a UnicodeEncodeError instance, which contains information about the location of the error. The error handler must either raise this or a different exception or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The encoder will encode the replacement and continue encoding the original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an IndexError will be raised.

Decoding and translating works similar, except UnicodeDecodeError or UnicodeTranslateError will be passed to the handler and that the replacement from the error handler will be put into the output directly.

codecs.lookup_error(name)
Return the error handler previously registered under the name name.

Raises a LookupError in case the handler cannot be found.

codecs.strict_errors(exception)
Implements the strict error handling: each encoding or decoding error raises a UnicodeError.

codecs.replace_errors(exception)
Implements the replace error handling: malformed data is replaced with a suitable replacement character such as '?' in bytestrings and '\ufffd' in Unicode strings.

codecs.ignore_errors(exception)
Implements the ignore error handling: malformed data is ignored and encoding or decoding is continued without further notice.

codecs.xmlcharrefreplace_errors(exception)
Implements the xmlcharrefreplace error handling (for encoding only): the unencodable character is replaced by an appropriate XML character reference.

codecs.backslashreplace_errors(exception)
Implements the backslashreplace error handling (for encoding only): the unencodable character is replaced by a backslashed escape sequence.

To simplify working with encoded files or stream, the module also defines these utility functions:

codecs.open(filename, mode[, encoding[, errors[, buffering]]])
Open an encoded file using the given mode and return a wrapped version providing transparent encoding/decoding. The default file mode is 'r' meaning to open the file in read mode.

Note: The wrapped version will only accept the object format defined by the codecs, i.e. Unicode objects for most built-in codecs. Output is also codec-dependent and will usually be Unicode as well.

Note: Files are always opened in binary mode, even if no binary mode was specified. This is done to avoid data loss due to encodings using 8-bit values. This means that no automatic conversion of '\n' is done on reading and writing.

---

encoding specifies the encoding which is to be used for the file.

errors may be given to define the error handling. It defaults to 'strict' which causes a ValueError to be raised in case an encoding error occurs.

buffering has the same meaning as for the built-in open() function. It defaults to line buffered.

codecs.EncodedFile(file, input[, output[, errors]])
Return a wrapped version of file which provides transparent encoding translation.

Strings written to the wrapped file are interpreted according to the given input encoding and then written to the original file as strings using the output encoding. The intermediate encoding will usually be Unicode but depends on the specified codecs.

If output is not given, it defaults to input.

errors may be given to define the error handling. It defaults to 'strict', which causes ValueError to be raised in case an encoding error occurs.

codecs.iterencode(iterable, encoding[, errors])
Uses an incremental encoder to iteratively encode the input provided by iterable. This function is a generator. errors (as well as any other keyword argument) is passed through to the incremental encoder.

New in version 2.5.

codecs.iterdecode(iterable, encoding[, errors])
: Uses an incremental decoder to iteratively decode the input provided by iterable. This function is a generator. errors (as well as any other keyword argument) is passed through to the incremental decoder.

  New in version 2.5.

The module also provides the following constants which are useful for reading and writing to platform dependent files:

codecs.BOM
codecs.BOM_BE
codecs.BOM_LE
codecs.BOM_UTF8
codecs.BOM_UTF16
codecs.BOM_UTF16_BE
codecs.BOM_UTF16_LE
codecs.BOM_UTF32
codecs.BOM_UTF32_BE
codecs.BOM_UTF32_LE
: These constants define various encodings of the Unicode byte order mark (BOM) used in UTF-16 and UTF-32 data streams to indicate the byte order used in the stream or file and in UTF-8 as a Unicode signature. BOM_UTF16 is either BOM_UTF16_BE or BOM_UTF16_LE depending on the platform's native byte order, BOM is an alias for BOM_UTF16, BOM_LE for BOM_UTF16_LE and BOM_BE for BOM_UTF16_BE. The others represent the BOM in UTF-8 and UTF-32 encodings.

7.8.1 Codec Base Classes

The codecs module defines a set of base classes which define the interface and can also be used to easily write your own codecs for use in Python.

Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols.

The Codec class defines the interface for stateless encoders/decoders.

To simplify and standardize error handling, the encode() and decode() methods may implement different error handling schemes by providing the errors string argument. The following string values are defined and implemented by all standard Python codecs:

Value	Meaning
'strict'	Raise UnicodeError (or a subclass); this is the default.
'ignore'	Ignore the character and continue with the next.
'replace'	Replace with a suitable replacement character; Python will use the official U+FFFD REPLACEMENT CHARACTER for the built-in Unicode codecs on decoding and '?' on encoding.
'xmlcharrefreplace'	Replace with the appropriate XML character reference (only for encoding).
'backslashreplace'	Replace with backslashed escape sequences (only for encoding).

The set of allowed values can be extended via register_error().

Codec Objects

The Codec class defines these methods which also define the function interfaces of the stateless encoder and decoder:

Codec.encode(input[, errors])
: Encodes the object input and returns a tuple (output object, length consumed). While codecs are not restricted to use with Unicode, in a Unicode context, encoding converts a Unicode object to a plain string using a particular character set encoding (e.g., cp1252 or iso-8859-1).

  errors defines the error handling to apply. It defaults to `'strict'` handling.

  The method may not store state in the Codec instance. Use StreamWriter for codecs which have to keep state in order to make encoding efficient.

  The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

Codec.decode(input[, errors])
: Decodes the object input and returns a tuple (output object, length consumed). In a Unicode context, decoding converts a plain string encoded using a particular character set encoding to a Unicode object.

  input must be an object which provides the bf_getreadbuf buffer slot. Python strings, buffer objects and memory mapped files are examples of objects providing this slot.

  errors defines the error handling to apply. It defaults to `'strict'` handling.

  The method may not store state in the Codec instance. Use StreamReader for codecs which have to keep state in order to make decoding efficient.

  The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

The IncrementalEncoder and IncrementalDecoder classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn't done with one call to the stateless encoder/decoder function, but with multiple calls to the encode()/decode() method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the encoding/decoding process during method calls.

The joined output of calls to the encode()/decode() method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.

IncrementalEncoder Objects

New in version 2.5.

The IncrementalEncoder class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

class codecs.IncrementalEncoder([errors])
: Constructor for an IncrementalEncoder instance.

  All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

  The IncrementalEncoder may implement different error handling schemes by providing the errors keyword argument. These parameters are predefined:

  - `'strict'` Raise ValueError (or a subclass); this is the default.
  - `'ignore'` Ignore the character and continue with the next.

- `'replace'` Replace with a suitable replacement character
- `'xmlcharrefreplace'` Replace with the appropriate XML character reference
- `'backslashreplace'` Replace with backslashed escape sequences.

The errors argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the IncrementalEncoder object.

The set of allowed values for the errors argument can be extended with register_error().

encode(object[, final])
: Encodes object (taking the current state of the encoder into account) and returns the resulting encoded object. If this is the last call to encode() final must be true (the default is false).

reset()
: Reset the encoder to the initial state.

IncrementalDecoder Objects

The IncrementalDecoder class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

class codecs.IncrementalDecoder([errors])
: Constructor for an IncrementalDecoder instance.

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The IncrementalDecoder may implement different error handling schemes by providing the errors keyword argument. These parameters are predefined:

- `'strict'` Raise ValueError (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character.

The errors argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the IncrementalDecoder object.

The set of allowed values for the errors argument can be extended with register_error().

decode(object[, final])
: Decodes object (taking the current state of the decoder into account) and returns the resulting decoded object. If this is the last call to decode() final must be true (the default is false). If final is true the decoder must decode the input completely and must flush all buffers. If this isn't possible (e.g. because of incomplete byte sequences at the end of the input) it must initiate error handling just like in the stateless case (which might raise an exception).

reset()
: Reset the decoder to the initial state.

The StreamWriter and StreamReader classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See encodings.utf_8 for an example of how this is done.

StreamWriter Objects

The StreamWriter class is a subclass of Codec and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

class codecs.StreamWriter(stream[, errors])
Constructor for a StreamWriter instance.

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

stream must be a file-like object open for writing binary data.

The StreamWriter may implement different error handling schemes by providing the errors keyword argument. These parameters are predefined:

- `'strict'` Raise ValueError (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character
- `'xmlcharrefreplace'` Replace with the appropriate XML character reference
- `'backslashreplace'` Replace with backslashed escape sequences.

The errors argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the StreamWriter object.

The set of allowed values for the errors argument can be extended with register_error().

write(object)
Writes the object's contents encoded to the stream.

writelines(list)
Writes the concatenated list of strings to the stream (possibly by reusing the write() method).

reset()
Flushes and resets the codec buffers used for keeping state.

Calling this method should ensure that the data on the output is put into a clean state that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the StreamWriter must also inherit all other methods and attributes from the underlying stream.

StreamReader Objects

The StreamReader class is a subclass of Codec and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.

class codecs.StreamReader(stream[, errors])
Constructor for a StreamReader instance.

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

stream must be a file-like object open for reading (binary) data.

The StreamReader may implement different error handling schemes by providing the errors keyword argument. These parameters are defined:

- `'strict'` Raise ValueError (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character.

The errors argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the StreamReader object.

The set of allowed values for the errors argument can be extended with register_error().

read([size[, chars[, firstline]]])
Decodes data from the stream and returns the resulting object.

chars indicates the number of characters to read from the stream. read() will never return more than chars characters, but it might return less, if there are not enough characters available.

size indicates the approximate maximum number of bytes to read from the stream for decoding purposes. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. size is intended to prevent having to decode huge files in one step.

firstline indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

Changed in version 2.4: chars argument added.

Changed in version 2.4.2: firstline argument added.

readline([size[, keepends]])
Read one line from the input stream and return the decoded data.

size, if given, is passed as size argument to the stream's read() method.

If keepends is false line-endings will be stripped from the lines returned.

Changed in version 2.4: keepends argument added.

readlines([sizehint[, keepends]])
Read all lines available on the input stream and return them as a list of lines.

Line-endings are implemented using the codec's decoder method and are included in the list entries if keepends is true.

sizehint, if given, is passed as the size argument to the stream's read() method.

reset()
Resets the codec buffers used for keeping state.

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the StreamReader must also inherit all other methods and attributes from the underlying stream.

The next two base classes are included for convenience. They are not needed by the codec registry, but may provide useful in practice.

StreamReaderWriter Objects

The StreamReaderWriter allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the lookup() function to construct the instance.

class codecs.StreamReaderWriter(stream, Reader, Writer, errors)
: Creates a StreamReaderWriter instance. stream must be a file-like object. Reader and Writer must be factory functions or classes providing the StreamReader and StreamWriter interface resp. Error handling is done in the same way as defined for the stream readers and writers.

StreamReaderWriter instances define the combined interfaces of StreamReader and StreamWriter classes. They inherit all other methods and attributes from the underlying stream.

StreamRecoder Objects

The StreamRecoder provide a frontend - backend view of encoding data which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the lookup() function to construct the instance.

class codecs.StreamRecoder(stream, encode, decode, Reader, Writer, errors)
: Creates a StreamRecoder instance which implements a two-way conversion: encode and decode work on the frontend (the input to read() and output of write()) while Reader and Writer work on the backend (reading and writing to the stream).

  You can use these objects to do transparent direct recodings from e.g. Latin-1 to UTF-8 and back.

  stream must be a file-like object.

  encode, decode must adhere to the Codec interface. Reader, Writer must be factory functions or classes providing objects of the StreamReader and StreamWriter interface respectively.

  encode and decode are needed for the frontend translation, Reader and Writer for the backend translation. The intermediate format used is determined by the two sets of codecs, e.g. the Unicode codecs will use Unicode as the intermediate encoding.

  Error handling is done in the same way as defined for the stream readers and writers.

StreamRecoder instances define the combined interfaces of StreamReader and StreamWriter classes. They inherit all other methods and attributes from the underlying stream.

7.8.2 Encodings and Unicode

Unicode strings are stored internally as sequences of code points (to be precise as Py_UNICODE arrays). Depending on the way Python is compiled (either via --enable-unicode=ucs2 or --enable-unicode=ucs4, with the former being the default) Py_UNICODE is either a 16-bit or 32-bit data type. Once a Unicode object is used outside of CPU and memory, CPU endianness and how these arrays are stored as bytes become an issue. Transforming a unicode object into a sequence of bytes is called encoding and recreating the unicode object from the sequence of bytes is known as decoding. There are many different methods for how this transformation can be done (these methods are also called encodings). The simplest method is to map the code points 0–255 to the bytes 0x0–0xff. This means that a unicode object that contains code points above U+00FF can't be encoded with this method (which is called 'latin-1' or 'iso-8859-1'). unicode.encode() will raise a UnicodeEncodeError that looks like this: UnicodeEncodeError: 'latin-1' codec can't encode character u'\u1234' in position 3: ordinal not in range(256).

There's another group of encodings (the so called charmap encodings) that choose a different subset of all unicode code points and how these code points are mapped to the bytes 0x0–0xff. To see how this is done simply open e.g. encodings/cp1252.py (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 code points defined in unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities: store the bytes in big endian or in little endian order. These two encodings are called UTF-32-BE and UTF-32-LE respectively. Their disadvantage is that if e.g. you use UTF-32-BE on a little endian machine you will always have to swap bytes on encoding and decoding. UTF-32 avoids this problem: bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a UTF-16 or UTF-32 byte sequence, there's the so called BOM ("Byte Order Mark"). This is the Unicode character U+FEFF. This character can be prepended to every UTF-16 or UTF-32 byte sequence. The byte swapped version of this character (0xFFFE) is an illegal character that may not appear in a Unicode text. So when the first character in an UTF-16 or UTF-32 byte sequence appears to be a U+FFFE the bytes have to be swapped on decoding. Unfortunately the character U+FEFF had a second purpose as a ZERO WIDTH NO-BREAK SPACE: a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using U+FEFF as a ZERO WIDTH NO-BREAK SPACE has been deprecated (with U+2060 (WORD JOINER) assuming this role). Nevertheless Unicode software still must be able to handle U+FEFF in both roles: as a BOM it's a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a Unicode string; as a ZERO WIDTH NO-BREAK SPACE it's a normal character that will be decoded like any other.

There's another encoding that is able to encoding the full range of Unicode characters: UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts: marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to four 1 bits followed by a 0 bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character):

Range	Encoding
U-00000000 ... U-0000007F	0xxxxxxx
U-00000080 ... U-000007FF	110xxxxx 10xxxxxx
U-00000800 ... U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 ... U-0010FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any U+FEFF character in the decoded Unicode string (even if it's the first character) is treated as a ZERO WIDTH NO-BREAK SPACE.

Without external information it's impossible to reliably determine which encoding was used for encoding a Unicode string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python 2.5 calls "utf-8-sig") for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: 0xef, 0xbb, 0xbf) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

> LATIN SMALL LETTER I WITH DIAERESIS
> RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
> INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a utf-8-sig encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write

0xef, 0xbb, 0xbf as the first three bytes to the file. On decoding utf-8-sig will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.

7.8.3 Standard Encodings

Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases; therefore, e.g. `'utf-8'` is a valid alias for the `'utf_8'` codec.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist:

- an ISO 8859 codeset
- a Microsoft Windows code page, which is typically derived from an 8859 codeset, but replaces control characters with additional graphic characters
- an IBM EBCDIC code page
- an IBM PC code page, which is ASCII compatible

Codec	Aliases	Languages
ascii	646, us-ascii	English
big5	big5-tw, csbig5	Traditional Chinese
big5hkscs	big5-hkscs, hkscs	Traditional Chinese
cp037	IBM037, IBM039	English
cp424	EBCDIC-CP-HE, IBM424	Hebrew
cp437	437, IBM437	English
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Western Europe
cp720		Arabic
cp737		Greek
cp775	IBM775	Baltic languages
cp850	850, IBM850	Western Europe
cp852	852, IBM852	Central and Eastern Europe
cp855	855, IBM855	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp856		Hebrew
cp857	857, IBM857	Turkish
cp858	858, IBM858	Western Europe
cp860	860, IBM860	Portuguese
cp861	861, CP-IS, IBM861	Icelandic
cp862	862, IBM862	Hebrew
cp863	863, IBM863	Canadian
cp864	IBM864	Arabic
cp865	865, IBM865	Danish, Norwegian
cp866	866, IBM866	Russian
cp869	869, CP-GR, IBM869	Greek
cp874		Thai

Continued on next page

Table 3 – continued from previous page

Codec	Aliases	Languages
cp875		Greek
cp932	932, ms932, mskanji, ms-kanji	Japanese
cp949	949, ms949, uhc	Korean
cp950	950, ms950	Traditional Chinese
cp1006		Urdu
cp1026	ibm1026	Turkish
cp1140	ibm1140	Western Europe
cp1250	windows-1250	Central and Eastern Europe
cp1251	windows-1251	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp1252	windows-1252	Western Europe
cp1253	windows-1253	Greek
cp1254	windows-1254	Turkish
cp1255	windows-1255	Hebrew
cp1256	windows-1256	Arabic
cp1257	windows-1257	Baltic languages
cp1258	windows-1258	Vietnamese
euc_jp	eucjp, ujis, u-jis	Japanese
euc_jis_2004	jisx0213, eucjis2004	Japanese
euc_jisx0213	eucjisx0213	Japanese
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	Korean
gb2312	chinese, csiso58gb231280, euc- cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	Simplified Chinese
gbk	936, cp936, ms936	Unified Chinese
gb18030	gb18030-2000	Unified Chinese
hz	hzgb, hz-gb, hz-gb-2312	Simplified Chinese
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	Japanese
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	Japanese
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	Japanese, Korean, Simplified Chinese, Western Europe, Greek
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	Japanese
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	Japanese
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	Japanese
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	Korean
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	West Europe
iso8859_2	iso-8859-2, latin2, L2	Central and Eastern Europe
iso8859_3	iso-8859-3, latin3, L3	Esperanto, Maltese
iso8859_4	iso-8859-4, latin4, L4	Baltic languages
iso8859_5	iso-8859-5, cyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
iso8859_6	iso-8859-6, arabic	Arabic
iso8859_7	iso-8859-7, greek, greek8	Greek
iso8859_8	iso-8859-8, hebrew	Hebrew

Continued on next page

Table 3 – continued from previous page

Codec	Aliases	Languages
iso8859_9	iso-8859-9, latin5, L5	Turkish
iso8859_10	iso-8859-10, latin6, L6	Nordic languages
iso8859_11	iso-8859-11, thai	Thai languages
iso8859_13	iso-8859-13, latin7, L7	Baltic languages
iso8859_14	iso-8859-14, latin8, L8	Celtic languages
iso8859_15	iso-8859-15, latin9, L9	Western Europe
iso8859_16	iso-8859-16, latin10, L10	South-Eastern Europe
johab	cp1361, ms1361	Korean
koi8_r		Russian
koi8_u		Ukrainian
mac_cyrillic	maccyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
mac_greek	macgreek	Greek
mac_iceland	maciceland	Icelandic
mac_latin2	maclatin2, maccentraleurope	Central and Eastern Europe
mac_roman	macroman	Western Europe
mac_turkish	macturkish	Turkish
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	Kazakh
shift_jis	csshiftjis, shiftjis, sjis, s_jis	Japanese
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	Japanese
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	Japanese
utf_32	U32, utf32	all languages
utf_32_be	UTF-32BE	all languages
utf_32_le	UTF-32LE	all languages
utf_16	U16, utf16	all languages
utf_16_be	UTF-16BE	all languages (BMP only)
utf_16_le	UTF-16LE	all languages (BMP only)
utf_7	U7, unicode-1-1-utf-7	all languages
utf_8	U8, UTF, utf8	all languages
utf_8_sig		all languages

7.8.4 Python Specific Encodings

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated purpose describes the encoding direction.

The following codecs provide unicode-to-str encoding[1] and str-to-unicode decoding[2], similar to the Unicode text encodings.

[1] str objects are also accepted as input in place of unicode objects. They are implicitly converted to unicode by decoding them using the default encoding. If this conversion fails, it may lead to encoding operations raising UnicodeDecodeError.

[2] unicode objects are also accepted as input in place of str objects. They are implicitly converted to str by encoding them using the default encoding. If this conversion fails, it may lead to decoding operations raising UnicodeEncodeError.

Codec	Aliases	Purpose
idna		Implements RFC 3490, see also encodings.idna
mbcs	dbcs	Windows only: Encode operand according to the ANSI codepage (CP_ACP)
palmos		Encoding of PalmOS 3.5
punycode		Implements RFC 3492
raw_unicode_escape		Produce a string that is suitable as raw Unicode literal in Python source code
rot_13	rot13	Returns the Caesar-cypher encryption of the operand
undefined		Raise an exception for all conversions. Can be used as the system encoding if no automatic coercion between byte and Unicode strings is desired.
unicode_escape		Produce a string that is suitable as Unicode literal in Python source code
unicode_internal		Return the internal representation of the operand

New in version 2.3: The idna and punycode encodings.

The following codecs provide str-to-str encoding and decoding[2].

Codec	Aliases	Purpose	Encoder/decoder
base64_codec	base64, base-64	Convert operand to multiline MIME base64 (the result always includes a trailing '\n')	base64.encodestring(), base64.decodestring()
bz2_codec	bz2	Compress the operand using bz2	bz2.compress(), bz2.decompress()
hex_codec	hex	Convert operand to hexadecimal representation, with two digits per byte	binascii.b2a_hex(), binascii.a2b_hex()
quopri_codec	quopri, quoted-printable, quotedprintable	Convert operand to MIME quoted printable	quopri.encode() with quotetabs=True, quopri.decode()
string_escape		Produce a string that is suitable as string literal in Python source code	
uu_codec	uu	Convert the operand using uuencode	uu.encode(), uu.decode()
zlib_codec	zip, zlib	Compress the operand using gzip	zlib.compress(), zlib.decompress()

7.8.5 encodings.idna — Internationalized Domain Names in Applications

New in version 2.3.

This module implements RFC 3490 (Internationalized Domain Names in Applications) and RFC 3492 (Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the punycode encoding and stringprep.

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as www.Alliancefrançaise.nu) is converted into an ASCII-compatible encoding (ACE, such as www.xn--alliancefranaise-npb.nu). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP Host fields, and so on. This conversion is carried out in the application; if possible invisible to the user: The application should transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

Python supports this conversion in several ways: the idna codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in section 3.1 (1) of RFC 3490 and converting each label to ACE as required, and conversely separating an input byte string into labels based on the . separator and converting any ACE labels found into unicode. Furthermore, the socket module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function parameters, such as httplib and ftplib, accept Unicode host names (httplib then also transparently sends an IDNA hostname in the Host field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: Applications wishing to present such host names to the user should decode them to Unicode.

The module encodings.idna also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The nameprep functions can be used directly if desired.

encodings.idna.nameprep(label)
: Return the nameprepped version of label. The implementation currently assumes query strings, so AllowUnassigned is true.

encodings.idna.ToASCII(label)
: Convert a label to ASCII, as specified in RFC 3490. UseSTD3ASCIIRules is assumed to be false.

encodings.idna.ToUnicode(label)
: Convert a label to Unicode, as specified in RFC 3490.

7.8.6 encodings.utf_8_sig — UTF-8 codec with BOM signature

New in version 2.5.

This module implements a variant of the UTF-8 codec: On encoding a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). For decoding an optional UTF-8 encoded BOM at the start of the data will be skipped.

7.9 unicodedata — Unicode Database

This module provides access to the Unicode Character Database which defines character properties for all Unicode characters. The data in this database is based on the UnicodeData.txt file version 5.2.0 which is publicly available from ftp://ftp.unicode.org/.

The module uses the same names and symbols as defined by the UnicodeData File Format 5.2.0 (see http://www.unicode.org/reports/tr44/tr44-4.html). It defines the following functions:

unicodedata.lookup(name)
: Look up character by name. If a character with the given name is found, return the corresponding Unicode character. If not found, KeyError is raised.

unicodedata.name(unichr[, default])
: Returns the name assigned to the Unicode character unichr as a string. If no name is defined, default is returned, or, if not given, ValueError is raised.

unicodedata.decimal(unichr[, default])
: Returns the decimal value assigned to the Unicode character unichr as integer. If no such value is defined, default is returned, or, if not given, ValueError is raised.

unicodedata.digit(unichr[, default])
: Returns the digit value assigned to the Unicode character unichr as integer. If no such value is defined, default is returned, or, if not given, ValueError is raised.

unicodedata.numeric(unichr[, default])
: Returns the numeric value assigned to the Unicode character unichr as float. If no such value is defined, default is returned, or, if not given, ValueError is raised.

unicodedata.category(unichr)
: Returns the general category assigned to the Unicode character unichr as string.

unicodedata.bidirectional(unichr)
: Returns the bidirectional class assigned to the Unicode character unichr as string. If no such value is defined, an empty string is returned.

unicodedata.combining(unichr)
: Returns the canonical combining class assigned to the Unicode character unichr as integer. Returns 0 if no combining class is defined.

unicodedata.east_asian_width(unichr)
: Returns the east asian width assigned to the Unicode character unichr as string.

  New in version 2.4.

unicodedata.mirrored(unichr)
: Returns the mirrored property assigned to the Unicode character unichr as integer. Returns 1 if the character has been identified as a "mirrored" character in bidirectional text, 0 otherwise.

unicodedata.decomposition(unichr)
: Returns the character decomposition mapping assigned to the Unicode character unichr as string. An empty string is returned in case no such mapping is defined.

unicodedata.normalize(form, unistr)
: Return the normal form form for the Unicode string unistr. Valid values for form are 'NFC', 'NFKC', 'NFD', and 'NFKD'.

  The Unicode standard defines various normalization forms of a Unicode string, based on the definition of canonical equivalence and compatibility equivalence. In Unicode, several characters can be expressed in various way. For example, the character U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA) can also be expressed as the sequence U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA).

  For each character, there are two normal forms: normal form C and normal form D. Normal form D (NFD) is also known as canonical decomposition, and translates each character into its decomposed form. Normal form C (NFC) first applies a canonical decomposition, then composes pre-combined characters again.

  In addition to these two forms, there are two additional normal forms based on compatibility equivalence. In Unicode, certain characters are supported which normally would be unified with other characters. For example, U+2160 (ROMAN NUMERAL ONE) is really the same thing as U+0049 (LATIN CAPITAL LETTER I). However, it is supported in Unicode for compatibility with existing character sets (e.g. gb2312).

  The normal form KD (NFKD) will apply the compatibility decomposition, i.e. replace all compatibility characters with their equivalents. The normal form KC (NFKC) first applies the compatibility decomposition, followed by the canonical composition.

  Even if two unicode strings are normalized and look the same to a human reader, if one has combining characters and the other doesn't, they may not compare equal.

  New in version 2.3.

In addition, the module exposes the following constant:

unicodedata.unidata_version
: The version of the Unicode database used in this module.

  New in version 2.3.

unicodedata.ucd_3_2_0
: This is an object that has the same methods as the entire module, but uses the Unicode database version 3.2 instead, for applications that require this specific version of the Unicode database (such as IDNA).

  New in version 2.5.

Examples:

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
u'{'
>>> unicodedata.name(u'/')
'SOLIDUS'
>>> unicodedata.decimal(u'9')
9
>>> unicodedata.decimal(u'a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category(u'A') # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional(u'\u0660') # 'A'rabic, 'N'umber
'AN'
```

7.10 stringprep — Internet String Preparation

New in version 2.3.

When identifying things (such as host names) in the internet, it is often necessary to compare such identifications for "equality". Exactly how this comparison is executed may depend on the application domain, e.g. whether it should be case-insensitive or not. It may be also necessary to restrict the possible identifications, to allow only identifications consisting of "printable" characters.

RFC 3454 defines a procedure for "preparing" Unicode strings in internet protocols. Before passing strings onto the wire, they are processed with the preparation procedure, after which they have a certain normalized form. The RFC defines a set of tables, which can be combined into profiles. Each profile must define which tables it uses, and what other optional parts of the stringprep procedure are part of the profile. One example of a stringprep profile is nameprep, which is used for internationalized domain names.

The module stringprep only exposes the tables from RFC 3454. As these tables would be very large to represent them as dictionaries or lists, the module uses the Unicode character database internally. The module source code itself was generated using the mkstringprep.py utility.

As a result, these tables are exposed as functions, not as data structures. There are two kinds of tables in the RFC: sets and mappings. For a set, stringprep provides the "characteristic function", i.e. a function that returns true if the parameter is part of the set. For mappings, it provides the mapping function: given the key, it returns the associated value. Below is a list of all functions available in the module.

stringprep.in_table_a1(code)
: Determine whether code is in tableA.1 (Unassigned code points in Unicode 3.2).

stringprep.in_table_b1(code)
Determine whether code is in tableB.1 (Commonly mapped to nothing).

stringprep.map_table_b2(code)
Return the mapped value for code according to tableB.2 (Mapping for case-folding used with NFKC).

stringprep.map_table_b3(code)
Return the mapped value for code according to tableB.3 (Mapping for case-folding used with no normalization).

stringprep.in_table_c11(code)
Determine whether code is in tableC.1.1 (ASCII space characters).

stringprep.in_table_c12(code)
Determine whether code is in tableC.1.2 (Non-ASCII space characters).

stringprep.in_table_c11_c12(code)
Determine whether code is in tableC.1 (Space characters, union of C.1.1 and C.1.2).

stringprep.in_table_c21(code)
Determine whether code is in tableC.2.1 (ASCII control characters).

stringprep.in_table_c22(code)
Determine whether code is in tableC.2.2 (Non-ASCII control characters).

stringprep.in_table_c21_c22(code)
Determine whether code is in tableC.2 (Control characters, union of C.2.1 and C.2.2).

stringprep.in_table_c3(code)
Determine whether code is in tableC.3 (Private use).

stringprep.in_table_c4(code)
Determine whether code is in tableC.4 (Non-character code points).

stringprep.in_table_c5(code)
Determine whether code is in tableC.5 (Surrogate codes).

stringprep.in_table_c6(code)
Determine whether code is in tableC.6 (Inappropriate for plain text).

stringprep.in_table_c7(code)
Determine whether code is in tableC.7 (Inappropriate for canonical representation).

stringprep.in_table_c8(code)
Determine whether code is in tableC.8 (Change display properties or are deprecated).

stringprep.in_table_c9(code)
Determine whether code is in tableC.9 (Tagging characters).

stringprep.in_table_d1(code)
Determine whether code is in tableD.1 (Characters with bidirectional property “R” or “AL”).

stringprep.in_table_d2(code)
Determine whether code is in tableD.2 (Characters with bidirectional property “L”).

7.11 fpformat — Floating point conversions

Deprecated since version 2.6: The fpformat module has been removed in Python 3.

The fpformat module defines functions for dealing with floating point numbers representations in 100% pure Python.

Note: This module is unnecessary: everything here can be done using the % string interpolation operator described in the String Formatting Operations section.

The fpformat module defines the following functions and an exception:

fpformat.fix(x, digs)
Format x as [-]ddd.ddd with digs digits after the point and at least one digit before. If digs <= 0, the decimal point is suppressed.

x can be either a number or a string that looks like one. digs is an integer.

Return value is a string.

fpformat.sci(x, digs)
Format x as [-]d.dddE[+-]ddd with digs digits after the point and exactly one digit before. If digs <= 0, one digit is kept and the point is suppressed.

x can be either a real number, or a string that looks like one. digs is an integer.

Return value is a string.

exception fpformat.NotANumber
Exception raised when a string passed to fix() or sci() as the x parameter does not look like a number. This is a subclass of ValueError when the standard exceptions are strings. The exception value is the improperly formatted string that caused the exception to be raised.

Example:

```
>>> import fpformat
>>> fpformat.fix(1.23, 1)
'1.2'
```

CHAPTER

EIGHT

DATA TYPES

The modules described in this chapter provide a variety of specialized data types such as dates and times, fixed-type arrays, heap queues, synchronized queues, and sets.

Python also provides some built-in data types, in particular, dict, list, set (which along with frozenset, replaces the deprecated sets module), and tuple. The str class can be used to handle binary data and 8-bit text, and the unicode class to handle Unicode text.

The following modules are documented in this chapter:

8.1 datetime — Basic date and time types

New in version 2.3.

The datetime module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation. For related functionality, see also the time and calendar modules.

There are two kinds of date and time objects: "naive" and "aware".

An aware object has sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, to locate itself relative to other aware objects. An aware object is used to represent a specific moment in time that is not open to interpretation[1].

A naive object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it's up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring aware objects, datetime and time objects have an optional time zone information attribute, tzinfo, that can be set to an instance of a subclass of the abstract tzinfo class. These tzinfo objects capture information about the offset from UTC time, the time zone name, and whether Daylight Saving Time is in effect. Note that no concrete tzinfo classes are supplied by the datetime module. Supporting timezones at whatever level of detail is required is up to the application. The rules for time adjustment across the world are more political than rational, and there is no standard suitable for every application.

The datetime module exports the following constants:

datetime.MINYEAR
: The smallest year number allowed in a date or datetime object. MINYEAR is 1.

[1] If, that is, we ignore the effects of Relativity

datetime.MAXYEAR
The largest year number allowed in a date or datetime object. MAXYEAR is 9999.

See also:

Module calendar General calendar related functions.

Module time Time access and conversions.

8.1.1 Available Types

class datetime.date
An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: year, month, and day.

class datetime.time
An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds (there is no notion of "leap seconds" here). Attributes: hour, minute, second, microsecond, and tzinfo.

class datetime.datetime
A combination of a date and a time. Attributes: year, month, day, hour, minute, second, microsecond, and tzinfo.

class datetime.timedelta
A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.

class datetime.tzinfo
An abstract base class for time zone information objects. These are used by the datetime and time classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

Objects of these types are immutable.

Objects of the date type are always naive.

An object of type time or datetime may be naive or aware. A datetime object d is aware if d.tzinfo is not None and d.tzinfo.utcoffset(d) does not return None. If d.tzinfo is None, or if d.tzinfo is not None but d.tzinfo.utcoffset(d) returns None, d is naive. A time object t is aware if t.tzinfo is not None and t.tzinfo.utcoffset(None) does not return None. Otherwise, t is naive.

The distinction between naive and aware doesn't apply to timedelta objects.

Subclass relationships:

```
object
    timedelta
    tzinfo
    time
    date
        datetime
```

8.1.2 timedelta Objects

A timedelta object represents a duration, the difference between two dates or times.

class datetime.timedelta([days[, seconds[, microseconds[, milliseconds[, minutes[, hours[, weeks]]]]]]])

All arguments are optional and default to 0. Arguments may be ints, longs, or floats, and may be positive or negative.

Only days, seconds and microseconds are stored internally. Arguments are converted to those units:

- A millisecond is converted to 1000 microseconds.
- A minute is converted to 60 seconds.
- An hour is converted to 3600 seconds.
- A week is converted to 7 days.

and days, seconds and microseconds are then normalized so that the representation is unique, with

- 0 <= microseconds < 1000000
- 0 <= seconds < 3600*24 (the number of seconds in one day)
- -999999999 <= days <= 999999999

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

If the normalized value of days lies outside the indicated range, OverflowError is raised.

Note that normalization of negative values may be surprising at first. For example,

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Class attributes are:

timedelta.min
The most negative timedelta object, timedelta(-999999999).

timedelta.max
The most positive timedelta object, timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999).

timedelta.resolution
The smallest possible difference between non-equal timedelta objects, timedelta(microseconds=1).

Note that, because of normalization, timedelta.max > -timedelta.min. -timedelta.max is not representable as a timedelta object.

Instance attributes (read-only):

Attribute	Value
days	Between -999999999 and 999999999 inclusive
seconds	Between 0 and 86399 inclusive
microseconds	Between 0 and 999999 inclusive

Supported operations:

Operation	Result
t1 = t2 + t3	Sum of t2 and t3. Afterwards t1-t2 == t3 and t1-t3 == t2 are true. (1)
t1 = t2 - t3	Difference of t2 and t3. Afterwards t1 == t2 - t3 and t2 == t1 + t3 are true. (1)
t1 = t2 * i or t1 = i * t2	Delta multiplied by an integer or long. Afterwards t1 // i == t2 is true, provided i != 0.
	In general, t1 * i == t1 * (i-1) + t1 is true. (1)
t1 = t2 // i	The floor is computed and the remainder (if any) is thrown away. (3)
+t1	Returns a timedelta object with the same value. (2)
-t1	equivalent to timedelta(-t1.days, -t1.seconds, -t1.microseconds), and to t1* -1. (1)(4)
abs(t)	equivalent to +t when t.days >= 0, and to -t when t.days < 0. (2)
str(t)	Returns a string in the form [D day[s],][H]H:MM:SS[.UUUUUU], where D is negative for negative t. (5)
repr(t)	Returns a string in the form datetime.timedelta(D[, S[, U]]), where D is negative for negative t. (5)

Notes:

(1) This is exact, but may overflow.

(2) This is exact, and cannot overflow.

(3) Division by 0 raises ZeroDivisionError.

(4) -timedelta.max is not representable as a timedelta object.

(5) String representations of timedelta objects are normalized similarly to their internal representation. This leads to somewhat unusual results for negative timedeltas. For example:

```
>>> timedelta(hours=-5)
datetime.timedelta(-1, 68400)
>>> print(_)
-1 day, 19:00:00
```

In addition to the operations listed above timedelta objects support certain additions and subtractions with date and datetime objects (see below).

Comparisons of timedelta objects are supported with the timedelta object representing the smaller duration considered to be the smaller timedelta. In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a timedelta object is compared to an object of a different type, TypeError is raised unless the comparison is == or !=. The latter cases return False or True, respectively.

timedelta objects are hashable (usable as dictionary keys), support efficient pickling, and in Boolean contexts, a timedelta object is considered to be true if and only if it isn't equal to timedelta(0).

Instance methods:

timedelta.total_seconds()

Return the total number of seconds contained in the duration. Equivalent to (td.microseconds + (td.seconds + td.days * 24 * 3600) * 10**6) / 10**6 computed with true division enabled.

Note that for very large time intervals (greater than 270 years on most platforms) this method will lose microsecond accuracy.

New in version 2.7.

Example usage:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)  # adds up to 365 days
>>> year.total_seconds()
31536000.0
>>> year == another_year
True
>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(3285), 9)
>>> three_years = nine_years // 3;
>>> three_years, three_years.days // 365
(datetime.timedelta(1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years + year
True
```

8.1.3 date Objects

A date object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book Calendrical Calculations, where it's the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

class datetime.date(year, month, day)
: All arguments are required. Arguments may be ints or longs, in the following ranges:

  - MINYEAR <= year <= MAXYEAR
  - 1 <= month <= 12
  - 1 <= day <= number of days in the given month and year

  If an argument outside those ranges is given, ValueError is raised.

Other constructors, all class methods:

classmethod date.today()
: Return the current local date. This is equivalent to date.fromtimestamp(time.time()).

classmethod date.fromtimestamp(timestamp)
: Return the local date corresponding to the POSIX timestamp, such as is returned by time.time(). This may raise ValueError, if the timestamp is out of the range of values supported by the platform C localtime() function. It's common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by fromtimestamp().

classmethod date.fromordinal(ordinal)
: Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. ValueError is raised unless 1 <= ordinal <= date.max.toordinal(). For any date d, date.fromordinal(d.toordinal()) == d.

Class attributes:

date.min
: The earliest representable date, date(MINYEAR, 1, 1).

date.max
: The latest representable date, date(MAXYEAR, 12, 31).

date.resolution
: The smallest possible difference between non-equal date objects, timedelta(days=1).

Instance attributes (read-only):

date.year
: Between MINYEAR and MAXYEAR inclusive.

date.month
: Between 1 and 12 inclusive.

date.day
: Between 1 and the number of days in the given month of the given year.

Supported operations:

Operation	Result
date2 = date1 + timedelta	date2 is timedelta.days days removed from date1. (1)
date2 = date1 - timedelta	Computes date2 such that date2 + timedelta == date1. (2)
timedelta = date1 - date2	(3)
date1 < date2	date1 is considered less than date2 when date1 precedes date2 in time. (4)

Notes:

(1) date2 is moved forward in time if timedelta.days > 0, or backward if timedelta.days < 0. Afterward date2 - date1 == timedelta.days. timedelta.seconds and timedelta.microseconds are ignored. OverflowError is raised if date2.year would be smaller than MINYEAR or larger than MAXYEAR.

(2) This isn't quite equivalent to date1 + (-timedelta), because -timedelta in isolation can overflow in cases where date1 - timedelta does not. timedelta.seconds and timedelta.microseconds are ignored.

(3) This is exact, and cannot overflow. timedelta.seconds and timedelta.microseconds are 0, and date2 + timedelta == date1 after.

(4) In other words, date1 < date2 if and only if date1.toordinal() < date2.toordinal(). In order to stop comparison from falling back to the default scheme of comparing object addresses, date comparison normally raises TypeError if the other comparand isn't also a date object. However, NotImplemented is returned instead if the other comparand has a timetuple() attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a date object is compared to an object of a different type, TypeError is raised unless the comparison is == or !=. The latter cases return False or True, respectively.

Dates can be used as dictionary keys. In Boolean contexts, all date objects are considered to be true.

Instance methods:

date.replace(year, month, day)
: Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified. For example, if d == date(2002, 12, 31), then d.replace(day=26) == date(2002, 12, 26).

date.timetuple()
: Return a time.struct_time such as returned by time.localtime(). The hours, minutes and seconds are 0, and the DST flag is -1. d.timetuple() is equivalent to time.struct_time((d.year, d.month, d.day, 0,

0, 0, d.weekday(), yday, -1)), where yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 is the day number within the current year starting with 1 for January 1st.

date.toordinal()
Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any date object d, date.fromordinal(d.toordinal()) == d.

date.weekday()
Return the day of the week as an integer, where Monday is 0 and Sunday is 6. For example, date(2002, 12, 4).weekday() == 2, a Wednesday. See also isoweekday().

date.isoweekday()
Return the day of the week as an integer, where Monday is 1 and Sunday is 7. For example, date(2002, 12, 4).isoweekday() == 3, a Wednesday. See also weekday(), isocalendar().

date.isocalendar()
Return a 3-tuple, (ISO year, ISO week number, ISO weekday).

The ISO calendar is a widely used variant of the Gregorian calendar. See https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm for a good explanation.

The ISO year consists of 52 or 53 full weeks, and where a week starts on a Monday and ends on a Sunday. The first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday. This is called week number 1, and the ISO year of that Thursday is the same as its Gregorian year.

For example, 2004 begins on a Thursday, so the first week of ISO year 2004 begins on Monday, 29 Dec 2003 and ends on Sunday, 4 Jan 2004, so that date(2003, 12, 29).isocalendar() == (2004, 1, 1) and date(2004, 1, 4).isocalendar() == (2004, 1, 7).

date.isoformat()
Return a string representing the date in ISO 8601 format, 'YYYY-MM-DD'. For example, date(2002, 12, 4).isoformat() == '2002-12-04'.

date.__str__()
For a date d, str(d) is equivalent to d.isoformat().

date.ctime()
Return a string representing the date, for example date(2002, 12, 4).ctime() == 'Wed Dec 4 00:00:00 2002'. d.ctime() is equivalent to time.ctime(time.mktime(d.timetuple())) on platforms where the native C ctime() function (which time.ctime() invokes, but which date.ctime() does not invoke) conforms to the C standard.

date.strftime(format)
Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. For a complete list of formatting directives, see section strftime() and strptime() Behavior.

date.__format__(format)
Same as date.strftime(). This makes it possible to specify a format string for a date object when using str.format(). See section strftime() and strptime() Behavior.

Example of counting days to an event:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
```

(continues on next page)

(continued from previous page)

```
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

Example of working with date:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)
>>> t = d.timetuple()
>>> for i in t:
...     print i
2002               # year
3                  # month
11                 # day
0
0
0
0                  # weekday (0 = Monday)
70                 # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print i
2002               # ISO year
11                 # ISO week number
1                  # ISO day number ( 1 = Monday )
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'
```

8.1.4 datetime Objects

A datetime object is a single object containing all the information from a date object and a time object. Like a date object, datetime assumes the current Gregorian calendar extended in both directions; like a time object, datetime assumes there are exactly 3600*24 seconds in every day.

Constructor:

class datetime.datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]])

The year, month and day arguments are required. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments may be ints or longs, in the following ranges:

- MINYEAR <= year <= MAXYEAR

- 1 <= month <= 12
- 1 <= day <= number of days in the given month and year
- 0 <= hour < 24
- 0 <= minute < 60
- 0 <= second < 60
- 0 <= microsecond < 1000000

If an argument outside those ranges is given, ValueError is raised.

Other constructors, all class methods:

classmethod datetime.today()
: Return the current local datetime, with tzinfo None. This is equivalent to datetime.fromtimestamp(time.time()). See also now(), fromtimestamp().

classmethod datetime.now([tz])
: Return the current local date and time. If optional argument tz is None or not specified, this is like today(), but, if possible, supplies more precision than can be gotten from going through a time.time() timestamp (for example, this may be possible on platforms supplying the C gettimeofday() function).

  If tz is not None, it must be an instance of a tzinfo subclass, and the current date and time are converted to tz's time zone. In this case the result is equivalent to tz.fromutc(datetime.utcnow().replace(tzinfo=tz)). See also today(), utcnow().

classmethod datetime.utcnow()
: Return the current UTC date and time, with tzinfo None. This is like now(), but returns the current UTC date and time, as a naive datetime object. See also now().

classmethod datetime.fromtimestamp(timestamp[, tz])
: Return the local date and time corresponding to the POSIX timestamp, such as is returned by time.time(). If optional argument tz is None or not specified, the timestamp is converted to the platform's local date and time, and the returned datetime object is naive.

  If tz is not None, it must be an instance of a tzinfo subclass, and the timestamp is converted to tz's time zone. In this case the result is equivalent to tz.fromutc(datetime.utcfromtimestamp(timestamp).replace(tzinfo=tz)).

  fromtimestamp() may raise ValueError, if the timestamp is out of the range of values supported by the platform C localtime() or gmtime() functions. It's common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by fromtimestamp(), and then it's possible to have two timestamps differing by a second that yield identical datetime objects. See also utcfromtimestamp().

classmethod datetime.utcfromtimestamp(timestamp)
: Return the UTC datetime corresponding to the POSIX timestamp, with tzinfo None. This may raise ValueError, if the timestamp is out of the range of values supported by the platform C gmtime() function. It's common for this to be restricted to years in 1970 through 2038. See also fromtimestamp().

classmethod datetime.fromordinal(ordinal)
: Return the datetime corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. ValueError is raised unless 1 <= ordinal <= datetime.max.toordinal(). The hour, minute, second and microsecond of the result are all 0, and tzinfo is None.

classmethod datetime.combine(date, time)
: Return a new datetime object whose date components are equal to the given date object's, and whose time components and tzinfo attributes are equal to the given time object's. For any datetime object

d, d == datetime.combine(d.date(), d.timetz()). If date is a datetime object, its time components and tzinfo attributes are ignored.

classmethod datetime.strptime(date_string, format)
Return a datetime corresponding to date_string, parsed according to format. This is equivalent to datetime(*(time.strptime(date_string, format)[0:6])). ValueError is raised if the date_string and format can't be parsed by time.strptime() or if it returns a value which isn't a time tuple. For a complete list of formatting directives, see section strftime() and strptime() Behavior.

New in version 2.5.

Class attributes:

datetime.min
The earliest representable datetime, datetime(MINYEAR, 1, 1, tzinfo=None).

datetime.max
The latest representable datetime, datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None).

datetime.resolution
The smallest possible difference between non-equal datetime objects, timedelta(microseconds=1).

Instance attributes (read-only):

datetime.year
Between MINYEAR and MAXYEAR inclusive.

datetime.month
Between 1 and 12 inclusive.

datetime.day
Between 1 and the number of days in the given month of the given year.

datetime.hour
In range(24).

datetime.minute
In range(60).

datetime.second
In range(60).

datetime.microsecond
In range(1000000).

datetime.tzinfo
The object passed as the tzinfo argument to the datetime constructor, or None if none was passed.

Supported operations:

Operation	Result
datetime2 = datetime1 + timedelta	(1)
datetime2 = datetime1 - timedelta	(2)
timedelta = datetime1 - datetime2	(3)
datetime1 < datetime2	Compares datetime to datetime. (4)

(1) datetime2 is a duration of timedelta removed from datetime1, moving forward in time if timedelta.days > 0, or backward if timedelta.days < 0. The result has the same tzinfo attribute as the input datetime, and datetime2 - datetime1 == timedelta after. OverflowError is raised if datetime2.year would be smaller than MINYEAR or larger than MAXYEAR. Note that no time zone adjustments are done even if the input is an aware object.

(2) Computes the datetime2 such that datetime2 + timedelta == datetime1. As for addition, the result has the same tzinfo attribute as the input datetime, and no time zone adjustments are done even if the input is aware. This isn't quite equivalent to datetime1 + (-timedelta), because -timedelta in isolation can overflow in cases where datetime1 - timedelta does not.

(3) Subtraction of a datetime from a datetime is defined only if both operands are naive, or if both are aware. If one is aware and the other is naive, TypeError is raised.

If both are naive, or both are aware and have the same tzinfo attribute, the tzinfo attributes are ignored, and the result is a timedelta object t such that datetime2 + t == datetime1. No time zone adjustments are done in this case.

If both are aware and have different tzinfo attributes, a-b acts as if a and b were first converted to naive UTC datetimes first. The result is (a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset()) except that the implementation never overflows.

(4) datetime1 is considered less than datetime2 when datetime1 precedes datetime2 in time.

If one comparand is naive and the other is aware, TypeError is raised. If both comparands are aware, and have the same tzinfo attribute, the common tzinfo attribute is ignored and the base datetimes are compared. If both comparands are aware and have different tzinfo attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from self.utcoffset()).

Note: In order to stop comparison from falling back to the default scheme of comparing object addresses, datetime comparison normally raises TypeError if the other comparand isn't also a datetime object. However, NotImplemented is returned instead if the other comparand has a timetuple() attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a datetime object is compared to an object of a different type, TypeError is raised unless the comparison is == or !=. The latter cases return False or True, respectively.

---

datetime objects can be used as dictionary keys. In Boolean contexts, all datetime objects are considered to be true.

Instance methods:

datetime.date()
Return date object with same year, month and day.

datetime.time()
Return time object with same hour, minute, second and microsecond. tzinfo is None. See also method timetz().

datetime.timetz()
Return time object with same hour, minute, second, microsecond, and tzinfo attributes. See also method time().

datetime.replace([year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]]])
Return a datetime with the same attributes, except for those attributes given new values by whichever keyword arguments are specified. Note that tzinfo=None can be specified to create a naive datetime from an aware datetime with no conversion of date and time data.

datetime.astimezone(tz)
Return a datetime object with new tzinfo attribute tz, adjusting the date and time data so the result is the same UTC time as self, but in tz's local time.

tz must be an instance of a tzinfo subclass, and its utcoffset() and dst() methods must not return None. self must be aware (self.tzinfo must not be None, and self.utcoffset() must not return None).

If self.tzinfo is tz, self.astimezone(tz) is equal to self: no adjustment of date or time data is performed. Else the result is local time in time zone tz, representing the same UTC time as self: after astz = dt.astimezone(tz), astz - astz.utcoffset() will usually have the same date and time data as dt - dt.utcoffset(). The discussion of class tzinfo explains the cases at Daylight Saving Time transition boundaries where this cannot be achieved (an issue only if tz models both standard and daylight time).

If you merely want to attach a time zone object tz to a datetime dt without adjustment of date and time data, use dt.replace(tzinfo=tz). If you merely want to remove the time zone object from an aware datetime dt without conversion of date and time data, use dt.replace(tzinfo=None).

Note that the default tzinfo.fromutc() method can be overridden in a tzinfo subclass to affect the result returned by astimezone(). Ignoring error cases, astimezone() acts like:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

datetime.utcoffset()

If tzinfo is None, returns None, else returns self.tzinfo.utcoffset(self), and raises an exception if the latter doesn't return None, or a timedelta object representing a whole number of minutes with magnitude less than one day.

datetime.dst()

If tzinfo is None, returns None, else returns self.tzinfo.dst(self), and raises an exception if the latter doesn't return None, or a timedelta object representing a whole number of minutes with magnitude less than one day.

datetime.tzname()

If tzinfo is None, returns None, else returns self.tzinfo.tzname(self), raises an exception if the latter doesn't return None or a string object,

datetime.timetuple()

Return a time.struct_time such as returned by time.localtime(). d.timetuple() is equivalent to time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), yday, dst)), where yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 is the day number within the current year starting with 1 for January 1st. The tm_isdst flag of the result is set according to the dst() method: tzinfo is None or dst() returns None, tm_isdst is set to -1; else if dst() returns a non-zero value, tm_isdst is set to 1; else tm_isdst is set to 0.

datetime.utctimetuple()

If datetime instance d is naive, this is the same as d.timetuple() except that tm_isdst is forced to 0 regardless of what d.dst() returns. DST is never in effect for a UTC time.

If d is aware, d is normalized to UTC time, by subtracting d.utcoffset(), and a time.struct_time for the normalized time is returned. tm_isdst is forced to 0. Note that the result's tm_year member may be MINYEAR-1 or MAXYEAR+1, if d.year was MINYEAR or MAXYEAR and UTC adjustment spills over a year boundary.

datetime.toordinal()

Return the proleptic Gregorian ordinal of the date. The same as self.date().toordinal().

datetime.weekday()

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. The same as self.date().weekday(). See also isoweekday().

datetime.isoweekday()
Return the day of the week as an integer, where Monday is 1 and Sunday is 7. The same as self.date().isoweekday(). See also weekday(), isocalendar().

datetime.isocalendar()
Return a 3-tuple, (ISO year, ISO week number, ISO weekday). The same as self.date().isocalendar().

datetime.isoformat([sep])
Return a string representing the date and time in ISO 8601 format, YYYY-MM-DDTHH:MM:SS.mmmmmm or, if microsecond is 0, YYYY-MM-DDTHH:MM:SS

If utcoffset() does not return None, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM or, if microsecond is 0 YYYY-MM-DDTHH:MM:SS+HH:MM

The optional argument sep (default 'T') is a one-character separator, placed between the date and time portions of the result. For example,

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

datetime.__str__()
For a datetime instance d, str(d) is equivalent to d.isoformat(' ').

datetime.ctime()
Return a string representing the date and time, for example datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec 4 20:30:40 2002'. d.ctime() is equivalent to time.ctime(time.mktime(d.timetuple())) on platforms where the native C ctime() function (which time.ctime() invokes, but which datetime.ctime() does not invoke) conforms to the C standard.

datetime.strftime(format)
Return a string representing the date and time, controlled by an explicit format string. For a complete list of formatting directives, see section strftime() and strptime() Behavior.

datetime.__format__(format)
Same as datetime.strftime(). This makes it possible to specify a format string for a datetime object when using str.format(). See section strftime() and strptime() Behavior.

Examples of working with datetime objects:

```
>>> from datetime import datetime, date, time
>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)
>>> # Using datetime.now() or datetime.utcnow()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)
>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)
```

(continues on next page)

(continued from previous page)

```
>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print it
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None
>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print it
...
2006    # ISO year
47      # ISO week
2       # ISO weekday
>>> # Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day", "month", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

Using datetime with tzinfo:

```
>>> from datetime import timedelta, datetime, tzinfo
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1) + self.dst(dt)
...     def dst(self, dt):
...         # DST starts last Sunday in March
...         d = datetime(dt.year, 4, 1)   # ends last Sunday in October
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <=  dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self,dt):
...          return "GMT +1"
...
>>> class GMT2(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=2) + self.dst(dt)
...     def dst(self, dt):
...         d = datetime(dt.year, 4, 1)
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <=  dt.replace(tzinfo=None) < self.dstoff:
```

(continues on next page)

(continued from previous page)

```
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self,dt):
...         return "GMT +2"
...
>>> gmt1 = GMT1()
>>> # Daylight Saving Time
>>> dt1 = datetime(2006, 11, 21, 16, 30, tzinfo=gmt1)
>>> dt1.dst()
datetime.timedelta(0)
>>> dt1.utcoffset()
datetime.timedelta(0, 3600)
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=gmt1)
>>> dt2.dst()
datetime.timedelta(0, 3600)
>>> dt2.utcoffset()
datetime.timedelta(0, 7200)
>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(GMT2())
>>> dt3     # doctest: +ELLIPSIS
datetime.datetime(2006, 6, 14, 14, 0, tzinfo=<GMT2 object at 0x...>)
>>> dt2     # doctest: +ELLIPSIS
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=<GMT1 object at 0x...>)
>>> dt2.utctimetuple() == dt3.utctimetuple()
True
```

8.1.5 time Objects

A time object represents a (local) time of day, independent of any particular day, and subject to adjustment via a tzinfo object.

class datetime.time([hour[, minute[, second[, microsecond[, tzinfo]]]]])
: All arguments are optional. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments may be ints or longs, in the following ranges:

  - 0 <= hour < 24
  - 0 <= minute < 60
  - 0 <= second < 60
  - 0 <= microsecond < 1000000.

  If an argument outside those ranges is given, ValueError is raised. All default to 0 except tzinfo, which defaults to None.

Class attributes:

time.min
: The earliest representable time, time(0, 0, 0, 0).

time.max
: The latest representable time, time(23, 59, 59, 999999).

time.resolution
: The smallest possible difference between non-equal time objects, timedelta(microseconds=1), although note that arithmetic on time objects is not supported.

Instance attributes (read-only):

time.hour
: In range(24).

time.minute
: In range(60).

time.second
: In range(60).

time.microsecond
: In range(1000000).

time.tzinfo
: The object passed as the tzinfo argument to the time constructor, or None if none was passed.

Supported operations:

- comparison of time to time, where a is considered less than b when a precedes b in time. If one comparand is naive and the other is aware, TypeError is raised. If both comparands are aware, and have the same tzinfo attribute, the common tzinfo attribute is ignored and the base times are compared. If both comparands are aware and have different tzinfo attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from self.utcoffset()). In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a time object is compared to an object of a different type, TypeError is raised unless the comparison is == or !=. The latter cases return False or True, respectively.
- hash, use as dict key
- efficient pickling
- in Boolean contexts, a time object is considered to be true if and only if, after converting it to minutes and subtracting utcoffset() (or 0 if that's None), the result is non-zero.

Instance methods:

time.replace([hour[, minute[, second[, microsecond[, tzinfo]]]]])
: Return a time with the same value, except for those attributes given new values by whichever keyword arguments are specified. Note that tzinfo=None can be specified to create a naive time from an aware time, without conversion of the time data.

time.isoformat()
: Return a string representing the time in ISO 8601 format, HH:MM:SS.mmmmmm or, if self.microsecond is 0, HH:MM:SS If utcoffset() does not return None, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: HH:MM:SS.mmmmmm+HH:MM or, if self.microsecond is 0, HH:MM:SS+HH:MM

time.__str__()
: For a time t, str(t) is equivalent to t.isoformat().

time.strftime(format)
: Return a string representing the time, controlled by an explicit format string. For a complete list of formatting directives, see section strftime() and strptime() Behavior.

time.__format__(format)
: Same as time.strftime(). This makes it possible to specify a format string for a time object when using str.format(). See section strftime() and strptime() Behavior.

time.utcoffset()
: If tzinfo is None, returns None, else returns self.tzinfo.utcoffset(None), and raises an exception if the

latter doesn't return None or a timedelta object representing a whole number of minutes with magnitude less than one day.

time.dst()
: If tzinfo is None, returns None, else returns self.tzinfo.dst(None), and raises an exception if the latter doesn't return None, or a timedelta object representing a whole number of minutes with magnitude less than one day.

time.tzname()
: If tzinfo is None, returns None, else returns self.tzinfo.tzname(None), or raises an exception if the latter doesn't return None or a string object.

Example:

```
>>> from datetime import time, tzinfo, timedelta
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "Europe/Prague"
...
>>> t = time(12, 10, 30, tzinfo=GMT1())
>>> t                           # doctest: +ELLIPSIS
datetime.time(12, 10, 30, tzinfo=<GMT1 object at 0x...>)
>>> gmt = GMT1()
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'Europe/Prague'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 Europe/Prague'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.6 tzinfo Objects

class datetime.tzinfo
: This is an abstract base class, meaning that this class should not be instantiated directly. You need to derive a concrete subclass, and (at least) supply implementations of the standard tzinfo methods needed by the datetime methods you use. The datetime module does not supply any concrete subclasses of tzinfo.

  An instance of (a concrete subclass of) tzinfo can be passed to the constructors for datetime and time objects. The latter objects view their attributes as being in local time, and the tzinfo object supports methods revealing offset of local time from UTC, the name of the time zone, and DST offset, all relative to a date or time object passed to them.

  Special requirement for pickling: A tzinfo subclass must have an __init__() method that can be called with no arguments, else it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

  A concrete subclass of tzinfo may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware datetime objects. If in doubt, simply implement all of them.

tzinfo.utcoffset(self, dt)

Return offset of local time from UTC, in minutes east of UTC. If local time is west of UTC, this should be negative. Note that this is intended to be the total offset from UTC; for example, if a tzinfo object represents both time zone and DST adjustments, utcoffset() should return their sum. If the UTC offset isn't known, return None. Else the value returned must be a timedelta object specifying a whole number of minutes in the range -1439 to 1439 inclusive (1440 = 24*60; the magnitude of the offset must be less than one day). Most implementations of utcoffset() will probably look like one of these two:

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

If utcoffset() does not return None, dst() should not return None either.

The default implementation of utcoffset() raises NotImplementedError.

tzinfo.dst(self, dt)

Return the daylight saving time (DST) adjustment, in minutes east of UTC, or None if DST information isn't known. Return timedelta(0) if DST is not in effect. If DST is in effect, return the offset as a timedelta object (see utcoffset() for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by utcoffset(), so there's no need to consult dst() unless you're interested in obtaining DST info separately. For example, datetime.timetuple() calls its tzinfo attribute's dst() method to determine how the tm_isdst flag should be set, and tzinfo.fromutc() calls dst() to account for DST changes when crossing time zones.

An instance tz of a tzinfo subclass that models both standard and daylight times must be consistent in this sense:

tz.utcoffset(dt) - tz.dst(dt)

must return the same result for every datetime dt with dt.tzinfo == tz For sane tzinfo subclasses, this expression yields the time zone's "standard offset", which should not depend on the date or the time, but only on geographic location. The implementation of datetime.astimezone() relies on this, but cannot detect violations; it's the programmer's responsibility to ensure it. If a tzinfo subclass cannot guarantee this, it may be able to override the default implementation of tzinfo.fromutc() to work correctly with astimezone() regardless.

Most implementations of dst() will probably look like one of these two:

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

or

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time.  Then

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

The default implementation of dst() raises NotImplementedError.

tzinfo.tzname(self, dt)

Return the time zone name corresponding to the datetime object dt, as a string. Nothing about

string names is defined by the datetime module, and there's no requirement that it mean anything in particular. For example, "GMT", "UTC", "-500", "-5:00", "EDT", "US/Eastern", "America/New York" are all valid replies. Return None if a string name isn't known. Note that this is a method rather than a fixed string primarily because some tzinfo subclasses will wish to return different names depending on the specific value of dt passed, especially if the tzinfo class is accounting for daylight time.

The default implementation of tzname() raises NotImplementedError.

These methods are called by a datetime or time object, in response to their methods of the same names. A datetime object passes itself as the argument, and a time object passes None as the argument. A tzinfo subclass's methods should therefore be prepared to accept a dt argument of None, or of class datetime.

When None is passed, it's up to the class designer to decide the best response. For example, returning None is appropriate if the class wishes to say that time objects don't participate in the tzinfo protocols. It may be more useful for utcoffset(None) to return the standard UTC offset, as there is no other convention for discovering the standard offset.

When a datetime object is passed in response to a datetime method, dt.tzinfo is the same object as self. tzinfo methods can rely on this, unless user code calls tzinfo methods directly. The intent is that the tzinfo methods interpret dt as being in local time, and not need worry about objects in other timezones.

There is one more tzinfo method that a subclass may wish to override:

tzinfo.fromutc(self, dt)

This is called from the default datetime.astimezone() implementation. When called from that, dt.tzinfo is self, and dt's date and time data are to be viewed as expressing a UTC time. The purpose of fromutc() is to adjust the date and time data, returning an equivalent datetime in self's local time.

Most tzinfo subclasses should be able to inherit the default fromutc() implementation without problems. It's strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default fromutc() implementation may not handle correctly in all cases is one where the standard offset (from UTC) depends on the specific date and time passed, which can happen for political reasons. The default implementations of astimezone() and fromutc() may not produce the result you want if the result is one of the hours straddling the moment the standard offset changes.

Skipping code for error cases, the default fromutc() implementation acts like:

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

Example tzinfo classes:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
```

(continues on next page)

(continued from previous page)

```
# A UTC class.

class UTC(tzinfo):
    """UTC"""

    def utcoffset(self, dt):
        return ZERO

    def tzname(self, dt):
        return "UTC"

    def dst(self, dt):
        return ZERO

utc = UTC()

# A class building tzinfo objects for fixed-offset time zones.
# Note that FixedOffset(0, "UTC") is a different way to build a
# UTC tzinfo object.

class FixedOffset(tzinfo):
    """Fixed offset in minutes east from UTC."""

    def __init__(self, offset, name):
        self.__offset = timedelta(minutes = offset)
        self.__name = name

    def utcoffset(self, dt):
        return self.__offset

    def tzname(self, dt):
        return self.__name

    def dst(self, dt):
        return ZERO

# A class capturing the platform's idea of local time.

import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET
```

(continues on next page)

(continued from previous page)

```
    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time; 1am standard time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 1)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time; 1am standard time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 1)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time;
# 1am standard time) on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

class USTimeZone(tzinfo):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self

        # Find start and end times for US DST. For years before 1967, return
        # ZERO for no DST.
        if 2006 < dt.year:
            dststart, dstend = DSTSTART_2007, DSTEND_2007
        elif 1986 < dt.year < 2007:
            dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
        elif 1966 < dt.year < 1987:
            dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
        else:
            return ZERO

        start = first_sunday_on_or_after(dststart.replace(year=dt.year))
        end = first_sunday_on_or_after(dstend.replace(year=dt.year))

        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        if start <= dt.replace(tzinfo=None) < end:
            return HOUR
        else:
            return ZERO

Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

Note that there are unavoidable subtleties twice per year in a tzinfo subclass accounting for both standard and daylight time, at the DST transition points. For concreteness, consider US Eastern (UTC -0500), where EDT begins the minute after 1:59 (EST) on the second Sunday in March, and ends the minute after 1:59

(EDT) on the first Sunday in November:

```
 UTC   3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
 EST  22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
 EDT  23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start  22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end  23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

When DST starts (the "start" line), the local wall clock leaps from 1:59 to 3:00. A wall time of the form 2:MM doesn't really make sense on that day, so astimezone(Eastern) won't deliver a result with hour == 2 on the day DST begins. In order for astimezone() to make this guarantee, the rzinfo.dst() method must consider times in the "missing hour" (2:MM for Eastern) to be in daylight time.

When DST ends (the "end" line), there's a potentially worse problem: there's an hour that can't be spelled unambiguously in local wall time: the last hour of daylight time. In Eastern, that's times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. astimezone() mimics the local clock's behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern. In order for astimezone() to make this guarantee, the tzinfo.dst() method must consider times in the "repeated hour" to be in standard time. This is easily arranged, as in the example, by expressing DST switch times in the time zone's standard local time.

Applications that can't bear such ambiguities should avoid using hybrid tzinfo subclasses; there are no ambiguities when using UTC, or any other fixed-offset tzinfo subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).

See also:

pytz The standard library has no tzinfo instances, but there exists a third-party library which brings the IANA timezone database (also known as the Olson database) to Python: pytz.

pytz contains up-to-date information and its usage is recommended.

IANA timezone database The Time Zone Database (often called tz or zoneinfo) contains code and data that represent the history of local time for many representative locations around the globe. It is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules.

8.1.7 strftime() and strptime() Behavior

date, datetime, and time objects all support a strftime(format) method, to create a string representing the time under the control of an explicit format string. Broadly speaking, d.strftime(fmt) acts like the time module's time.strftime(fmt, d.timetuple()) although not all objects support a timetuple() method.

Conversely, the datetime.strptime() class method creates a datetime object from a string representing a date and time and a corresponding format string. datetime.strptime(date_string, format) is equivalent to datetime(*(time.strptime(date_string, format)[0:6])), except when the format includes sub-second components or timezone offset information, which are supported in datetime.strptime but are discarded by time.strptime.

For time objects, the format codes for year, month, and day should not be used, as time objects have no such values. If they're used anyway, 1900 is substituted for the year, and 1 for the month and day.

For date objects, the format codes for hours, minutes, seconds, and microseconds should not be used, as date objects have no such values. If they're used anyway, 0 is substituted for them.

The full set of format codes supported varies across platforms, because Python calls the platform C library's strftime() function, and platform variations are common. To see the full set of format codes supported on your platform, consult the strftime(3) documentation.

For the same reason, handling of format strings containing Unicode code points that can't be represented in the charset of the current locale is also platform-dependent. On some platforms such code points are preserved intact in the output, while on others strftime may raise UnicodeError or return an empty string instead.

The following is a list of all the format codes that the C standard (1989 version) requires, and these work on all platforms with a standard C implementation. Note that the 1999 version of the C standard added additional format codes.

The exact range of years for which strftime() works also varies across platforms. Regardless of platform, years before 1900 cannot be used.

Directive	Meaning	Example	Notes
%a	Weekday as locale's abbreviated name.	Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)	(1)
%A	Weekday as locale's full name.	Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)	(1)
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, ..., 6	
%d	Day of the month as a zero-padded decimal number.	01, 02, ..., 31	
%b	Month as locale's abbreviated name.	Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)	(1)
%B	Month as locale's full name.	January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)	(1)
%m	Month as a zero-padded decimal number.	01, 02, ..., 12	
%y	Year without century as a zero-padded decimal number.	00, 01, ..., 99	
%Y	Year with century as a decimal number.	1970, 1988, 2001, 2013	
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23	
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12	
%p	Locale's equivalent of either AM or PM.	AM, PM (en_US); am, pm (de_DE)	(1), (2)
%M	Minute as a zero-padded decimal number.	00, 01, ..., 59	
%S	Second as a zero-padded decimal number.	00, 01, ..., 59	(3)
%f	Microsecond as a decimal number, zero-padded on the left.	000000, 000001, ..., 999999	(4)

Notes:

(1) Because the format depends on the current locale, care should be taken when making assumptions about the output value. Field orderings will vary (for example, "month/day/year" versus "day/month/year"), and the output may contain Unicode characters encoded using the locale's default encoding (for example, if the current locale is ja_JP, the default encoding could be any one of eucJP, SJIS, or utf-8; use locale.getlocale() to determine the current locale's encoding).

(2) When used with the strptime() method, the %p directive only affects the output hour field if the %I directive is used to parse the hour.

(3) Unlike the time module, the datetime module does not support leap seconds.

(4) %f is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available). When used with the strptime() method, the %f directive accepts from one to six digits and zero pads on the right.

New in version 2.6.

(5) For a naive object, the %z and %Z format codes are replaced by empty strings.

For an aware object:

%z utcoffset() is transformed into a 5-character string of the form +HHMM or -HHMM, where HH is a 2-digit string giving the number of UTC offset hours, and MM is a 2-digit string giving the number of UTC offset minutes. For example, if utcoffset() returns timedelta(hours=-3, minutes=-30), %z is replaced with the string `'-0330'`.

%Z If tzname() returns None, %Z is replaced by an empty string. Otherwise %Z is replaced by the returned value, which must be a string.

(6) When used with the strptime() method, %U and %W are only used in calculations when the day of the week and the year are specified.

8.2 calendar — General calendar-related functions

Source code: Lib/calendar.py

This module allows you to output calendars like the Unix cal program, and provides additional useful functions related to the calendar. By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention). Use setfirstweekday() to set the first day of the week to Sunday (6) or to any other weekday. Parameters that specify dates are given as integers. For related functionality, see also the datetime and time modules.

Most of these functions and classes rely on the datetime module which uses an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book "Calendrical Calculations", where it's the base calendar for all computations.

class calendar.Calendar([firstweekday])
Creates a Calendar object. firstweekday is an integer specifying the first day of the week. 0 is Monday (the default), 6 is Sunday.

A Calendar object provides several methods that can be used for preparing the calendar data for formatting. This class doesn't do any formatting itself. This is the job of subclasses.

New in version 2.5.

Calendar instances have the following methods:

iterweekdays()
: Return an iterator for the week day numbers that will be used for one week. The first value from the iterator will be the same as the value of the firstweekday property.

itermonthdates(year, month)
: Return an iterator for the month month (1–12) in the year year. This iterator will return all days (as datetime.date objects) for the month and all days before the start of the month or after the end of the month that are required to get a complete week.

itermonthdays2(year, month)
: Return an iterator for the month month in the year year similar to itermonthdates(). Days returned will be tuples consisting of a day number and a week day number.

itermonthdays(year, month)
: Return an iterator for the month month in the year year similar to itermonthdates(). Days returned will simply be day numbers.

monthdatescalendar(year, month)
: Return a list of the weeks in the month month of the year as full weeks. Weeks are lists of seven datetime.date objects.

monthdays2calendar(year, month)
: Return a list of the weeks in the month month of the year as full weeks. Weeks are lists of seven tuples of day numbers and weekday numbers.

monthdayscalendar(year, month)
: Return a list of the weeks in the month month of the year as full weeks. Weeks are lists of seven day numbers.

yeardatescalendar(year[, width])
: Return the data for the specified year ready for formatting. The return value is a list of month rows. Each month row contains up to width months (defaulting to 3). Each month contains between 4 and 6 weeks and each week contains 1–7 days. Days are datetime.date objects.

yeardays2calendar(year[, width])
: Return the data for the specified year ready for formatting (similar to yeardatescalendar()). Entries in the week lists are tuples of day numbers and weekday numbers. Day numbers outside this month are zero.

yeardayscalendar(year[, width])
: Return the data for the specified year ready for formatting (similar to yeardatescalendar()). Entries in the week lists are day numbers. Day numbers outside this month are zero.

class calendar.TextCalendar([firstweekday])
: This class can be used to generate plain text calendars.

  New in version 2.5.

  TextCalendar instances have the following methods:

  formatmonth(theyear, themonth[, w[, l]])
  : Return a month's calendar in a multi-line string. If w is provided, it specifies the width of the date columns, which are centered. If l is given, it specifies the number of lines that each week will use. Depends on the first weekday as specified in the constructor or set by the setfirstweekday() method.

  prmonth(theyear, themonth[, w[, l]])
  : Print a month's calendar as returned by formatmonth().

  formatyear(theyear[, w[, l[, c[, m]]]])
  : Return a m-column calendar for an entire year as a multi-line string. Optional parameters w, l,

and c are for date column width, lines per week, and number of spaces between month columns, respectively. Depends on the first weekday as specified in the constructor or set by the setfirstweekday() method. The earliest year for which a calendar can be generated is platform-dependent.

pryear(theyear[, w[, l[, c[, m]]]])
Print the calendar for an entire year as returned by formatyear().

class calendar.HTMLCalendar([firstweekday])
This class can be used to generate HTML calendars.

New in version 2.5.

HTMLCalendar instances have the following methods:

formatmonth(theyear, themonth[, withyear])
Return a month's calendar as an HTML table. If withyear is true the year will be included in the header, otherwise just the month name will be used.

formatyear(theyear[, width])
Return a year's calendar as an HTML table. width (defaulting to 3) specifies the number of months per row.

formatyearpage(theyear[, width[, css[, encoding]]])
Return a year's calendar as a complete HTML page. width (defaulting to 3) specifies the number of months per row. css is the name for the cascading style sheet to be used. None can be passed if no style sheet should be used. encoding specifies the encoding to be used for the output (defaulting to the system default encoding).

class calendar.LocaleTextCalendar([firstweekday[, locale]])
This subclass of TextCalendar can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

New in version 2.5.

class calendar.LocaleHTMLCalendar([firstweekday[, locale]])
This subclass of HTMLCalendar can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

New in version 2.5.

Note: The formatweekday() and formatmonthname() methods of these two classes temporarily change the current locale to the given locale. Because the current locale is a process-wide setting, they are not thread-safe.

For simple text calendars this module provides the following functions.

calendar.setfirstweekday(weekday)
Sets the weekday (0 is Monday, 6 is Sunday) to start each week. The values MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, and SUNDAY are provided for convenience. For example, to set the first weekday to Sunday:

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

New in version 2.0.

calendar.firstweekday()
Returns the current setting for the weekday to start each week.

New in version 2.0.

calendar.isleap(year)
Returns True if year is a leap year, otherwise False.

calendar.leapdays(y1, y2)
Returns the number of leap years in the range from y1 to y2 (exclusive), where y1 and y2 are years.

Changed in version 2.0: This function didn't work for ranges spanning a century change in Python 1.5.2.

calendar.weekday(year, month, day)
Returns the day of the week (0 is Monday) for year (1970–...), month (1–12), day (1–31).

calendar.weekheader(n)
Return a header containing abbreviated weekday names. n specifies the width in characters for one weekday.

calendar.monthrange(year, month)
Returns weekday of first day of the month and number of days in month, for the specified year and month.

calendar.monthcalendar(year, month)
Returns a matrix representing a month's calendar. Each row represents a week; days outside of the month a represented by zeros. Each week begins with Monday unless set by setfirstweekday().

calendar.prmonth(theyear, themonth[, w[, l]])
Prints a month's calendar as returned by month().

calendar.month(theyear, themonth[, w[, l]])
Returns a month's calendar in a multi-line string using the formatmonth() of the TextCalendar class.

New in version 2.0.

calendar.prcal(year[, w[, l[c]]])
Prints the calendar for an entire year as returned by calendar().

calendar.calendar(year[, w[, l[c]]])
Returns a 3-column calendar for an entire year as a multi-line string using the formatyear() of the TextCalendar class.

New in version 2.0.

calendar.timegm(tuple)
An unrelated but handy function that takes a time tuple such as returned by the gmtime() function in the time module, and returns the corresponding Unix timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, time.gmtime() and timegm() are each others' inverse.

New in version 2.0.

The calendar module exports the following data attributes:

calendar.day_name
An array that represents the days of the week in the current locale.

calendar.day_abbr
An array that represents the abbreviated days of the week in the current locale.

calendar.month_name
: An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and month_name[0] is the empty string.

calendar.month_abbr
: An array that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and month_abbr[0] is the empty string.

See also:

Module datetime Object-oriented interface to dates and times with similar functionality to the time module.

Module time Low-level time related functions.

8.3 collections — High-performance container datatypes

New in version 2.4.

Source code: Lib/collections.py and Lib/_abcoll.py

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple.

namedtuple()	factory function for creating tuple subclasses with named fields	New in version 2.6.
deque	list-like container with fast appends and pops on either end	New in version 2.4.
Counter	dict subclass for counting hashable objects	New in version 2.7.
OrderedDict	dict subclass that remembers the order entries were added	New in version 2.7.
defaultdict	dict subclass that calls a factory function to supply missing values	New in version 2.5.

In addition to the concrete container classes, the collections module provides abstract base classes that can be used to test whether a class provides a particular interface, for example, whether it is hashable or a mapping.

8.3.1 Counter objects

A counter tool is provided to support convenient and rapid tallies. For example:

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.Counter([iterable-or-mapping])

A Counter is a dict subclass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The Counter class is similar to bags or multisets in other languages.

Elements are counted from an iterable or initialized from another mapping (or counter):

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a KeyError:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

Setting a count to zero does not remove an element from a counter. Use del to remove it entirely:

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

New in version 2.7.

Counter objects support three methods beyond those available for all dictionaries:

elements()

Return an iterator over elements repeating each as many times as its count. Elements are returned in arbitrary order. If an element's count is less than one, elements() will ignore it.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([n])

Return a list of the n most common elements and their counts from the most common to the least. If n is omitted or None, most_common() returns all elements in the counter. Elements with equal counts are ordered arbitrarily:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

subtract([iterable-or-mapping])

Elements are subtracted from an iterable or from another mapping (or counter). Like dict.update() but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

The usual dictionary methods are available for Counter objects except for two which work differently for counters.

fromkeys(iterable)
: This class method is not implemented for Counter objects.

update([iterable-or-mapping])
: Elements are counted from an iterable or added-in from another mapping (or counter). Like dict.update() but adds counts instead of replacing them. Also, the iterable is expected to be a sequence of elements, not a sequence of (key, value) pairs.

Common patterns for working with Counter objects:

```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]       # n least common elements
c += Counter()                  # remove zero and negative counts
```

Several mathematical operations are provided for combining Counter objects to produce multisets (counters that have counts greater than zero). Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements. Intersection and union return the minimum and maximum of corresponding counts. Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                       # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                       # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

Note: Counters were primarily designed to work with positive integers to represent running counts; however, care was taken to not unnecessarily preclude use cases needing other types or negative values. To help with those use cases, this section documents the minimum range and type restrictions.

- The Counter class itself is a dictionary subclass with no restrictions on its keys and values. The values are intended to be numbers representing counts, but you could store anything in the value field.
- The most_common() method requires only that the values be orderable.
- For in-place operations such as c[key] += 1, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for update() and subtract() which allow negative and zero values for both inputs and outputs.
- The multiset methods are designed only for use cases with positive values. The inputs may be negative or zero, but only outputs with positive values are created. There are no type restrictions, but the value type needs to support addition, subtraction, and comparison.
- The elements() method requires integer counts. It ignores zero and negative counts.

See also:

- Counter class adapted for Python 2.5 and an early Bag recipe for Python 2.4.
- Bag class in Smalltalk.
- Wikipedia entry for Multisets.
- C++ multisets tutorial with examples.
- For mathematical operations on multisets and their use cases, see Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19.
- To enumerate all distinct multisets of a given size over a given set of elements, see itertools.combinations_with_replacement().

  map(Counter, combinations_with_replacement('ABC', 2)) –> AA AB AC BB BC CC

8.3.2 deque objects

class collections.deque([iterable[, maxlen]])

Returns a new deque object initialized left-to-right (using append()) with data from iterable. If iterable is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced "deck" and is short for "double-ended queue"). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.

New in version 2.4.

If maxlen is not specified or is None, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the tail filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Changed in version 2.6: Added maxlen parameter.

Deque objects support the following methods:

append(x)
: Add x to the right side of the deque.

appendleft(x)
: Add x to the left side of the deque.

clear()
: Remove all elements from the deque leaving it with length 0.

count(x)
: Count the number of deque elements equal to x.

  New in version 2.7.

extend(iterable)
: Extend the right side of the deque by appending elements from the iterable argument.

extendleft(iterable)
: Extend the left side of the deque by appending elements from iterable. Note, the series of left appends results in reversing the order of elements in the iterable argument.

pop()
Remove and return an element from the right side of the deque. If no elements are present, raises an IndexError.

popleft()
Remove and return an element from the left side of the deque. If no elements are present, raises an IndexError.

remove(value)
Remove the first occurrence of value. If not found, raises a ValueError.

New in version 2.5.

reverse()
Reverse the elements of the deque in-place and then return None.

New in version 2.7.

rotate(n=1)
Rotate the deque n steps to the right. If n is negative, rotate to the left.

When the deque is not empty, rotating one step to the right is equivalent to d.appendleft(d.pop()), and rotating one step to the left is equivalent to d.append(d.popleft()).

Deque objects also provide one read-only attribute:

maxlen
Maximum size of a deque or None if unbounded.

New in version 2.7.

In addition to the above, deques support iteration, pickling, len(d), reversed(d), copy.copy(d), copy.deepcopy(d), membership testing with the in operator, and subscript references such as d[-1]. Indexed access is O(1) at both ends but slows to O(n) in the middle. For fast random access, use lists instead.

Example:

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print elem.upper()
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
'i'
```

(continues on next page)

(continued from previous page)

```
>>> list(reversed(d))               # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                        # search the deque
True
>>> d.extend('jkl')                 # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                     # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                    # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))              # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                       # empty the deque
>>> d.pop()                         # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')             # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

deque Recipes

This section shows various approaches to working with deques.

Bounded length deques provide functionality similar to the tail filter in Unix:

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    return deque(open(filename), n)
```

Another approach to using deques is to maintain a sequence of recently added elements by appending to the right and popping to the left:

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / float(n)
```

The rotate() method provides a way to implement deque slicing and deletion. For example, a pure Python implementation of del d[n] relies on the rotate() method to position elements to be popped:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

To implement deque slicing, use a similar approach applying rotate() to bring a target element to the left side of the deque. Remove old entries with popleft(), add new entries with extend(), and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations such as dup, drop, swap, over, pick, rot, and roll.

8.3.3 defaultdict objects

class collections.defaultdict([default_factory[, ...]])

Returns a new dictionary-like object. defaultdict is a subclass of the built-in dict class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the dict class and is not documented here.

The first argument provides the initial value for the default_factory attribute; it defaults to None. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

New in version 2.5.

defaultdict objects support the following method in addition to the standard dict operations:

__missing__(key)

If the default_factory attribute is None, this raises a KeyError exception with the key as argument.

If default_factory is not None, it is called without arguments to provide a default value for the given key, this value is inserted in the dictionary for the key, and returned.

If calling default_factory raises an exception this exception is propagated unchanged.

This method is called by the __getitem__() method of the dict class when the requested key is not found; whatever it returns or raises is then returned or raised by __getitem__().

Note that __missing__() is not called for any operations besides __getitem__(). This means that get() will, like normal dictionaries, return None as a default rather than using default_factory.

defaultdict objects support the following instance variable:

default_factory

This attribute is used by the __missing__() method; it is initialized from the first argument to the constructor, if present, or to None, if absent.

defaultdict Examples

Using list as the default_factory, it is easy to group a sequence of key-value pairs into a dictionary of lists:

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the default_factory function which returns an empty list. The list.append() operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the list.append() operation adds another value to the list. This technique is simpler and faster than an equivalent technique using dict.setdefault():

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Setting the default_factory to int makes the defaultdict useful for counting (like a bag or multiset in other languages):

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> d.items()
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

When a letter is first encountered, it is missing from the mapping, so the default_factory function calls int() to supply a default count of zero. The increment operation then builds up the count for each letter.

The function int() which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use itertools.repeat() which can supply any constant value (not just zero):

```
>>> def constant_factory(value):
...     return itertools.repeat(value).next
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

Setting the default_factory to set makes the defaultdict useful for building a dictionary of sets:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> d.items()
[('blue', set([2, 4])), ('red', set([1, 3]))]
```

8.3.4 namedtuple() Factory Function for Tuples with Named Fields

Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

collections.namedtuple(typename, field_names[, verbose=False][, rename=False])

Returns a new tuple subclass named typename. The new subclass is used to create tuple-like objects

that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with typename and field_names) and a helpful __repr__() method which lists the tuple contents in a name=value format.

The field_names are a sequence of strings such as ['x', 'y']. Alternatively, field_names can be a single string with each fieldname separated by whitespace and/or commas, for example 'x y' or 'x, y'.

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a keyword such as class, for, return, global, pass, print, or raise.

If rename is true, invalid fieldnames are automatically replaced with positional names. For example, ['abc', 'def', 'ghi', 'abc'] is converted to ['abc', '_1', 'ghi', '_3'], eliminating the keyword def and the duplicate fieldname abc.

If verbose is true, the class definition is printed just before being built.

Named tuple instances do not have per-instance dictionaries, so they are lightweight and require no more memory than regular tuples.

New in version 2.6.

Changed in version 2.7: added support for rename.

Example:

```
>>> Point = namedtuple('Point', ['x', 'y'], verbose=True)
class Point(tuple):
    'Point(x, y)'

    __slots__ = ()

    _fields = ('x', 'y')

    def __new__(_cls, x, y):
        'Create new instance of Point(x, y)'
        return _tuple.__new__(_cls, (x, y))

    @classmethod
    def _make(cls, iterable, new=tuple.__new__, len=len):
        'Make a new Point object from a sequence or iterable'
        result = new(cls, iterable)
        if len(result) != 2:
            raise TypeError('Expected 2 arguments, got %d' % len(result))
        return result

    def __repr__(self):
        'Return a nicely formatted representation string'
        return 'Point(x=%r, y=%r)' % self

    def _asdict(self):
        'Return a new OrderedDict which maps field names to their values'
        return OrderedDict(zip(self._fields, self))

    def _replace(_self, **kwds):
        'Return a new Point object replacing specified fields with new values'
        result = _self._make(map(kwds.pop, ('x', 'y'), _self))
        if kwds:
            raise ValueError('Got unexpected field names: %r' % kwds.keys())
```

(continues on next page)

(continued from previous page)

```
        return result

    def __getnewargs__(self):
        'Return self as a plain tuple.  Used by copy and pickle.'
        return tuple(self)

    __dict__ = _property(_asdict)

    def __getstate__(self):
        'Exclude the OrderedDict from pickling'
        pass

    x = _property(_itemgetter(0), doc='Alias for field number 0')

    y = _property(_itemgetter(1), doc='Alias for field number 1')



>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y               # fields also accessible by name
33
>>> p                       # readable __repr__ with a name=value style
Point(x=11, y=22)
```

Named tuples are especially useful for assigning field names to result tuples returned by the csv or sqlite3 modules:

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print emp.name, emp.title

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print emp.name, emp.title
```

In addition to the methods inherited from tuples, named tuples support three additional methods and one attribute. To prevent conflicts with field names, the method and attribute names start with an underscore.

classmethod somenamedtuple._make(iterable)
: Class method that makes a new instance from an existing sequence or iterable.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple._asdict()
: Return a new OrderedDict which maps field names to their corresponding values:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

Changed in version 2.7: Returns an OrderedDict instead of a regular dict.

somenamedtuple._replace(**kwargs)

Return a new instance of the named tuple replacing specified fields with new values:

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum], timestamp=time.now())
```

somenamedtuple._fields

Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

To retrieve a field whose name is stored in a string, use the getattr() function:

```
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the double-star-operator (as described in tut-unpacking-arguments):

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Since a named tuple is a regular Python class, it is easy to add or change functionality with a subclass. Here is how to add a calculated field and a fixed-width print format:

```
>>> class Point(namedtuple('Point', 'x y')):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.hypot)
...
>>> for p in Point(3, 4), Point(14, 5/7.):
...     print p
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

The subclass shown above sets __slots__ to an empty tuple. This helps keep memory requirements low by preventing the creation of instance dictionaries.

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the _fields attribute:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Default values can be implemented by using _replace() to customize a prototype instance:

```
>>> Account = namedtuple('Account', 'owner balance transaction_count')
>>> default_account = Account('<owner name>', 0.0, 0)
>>> johns_account = default_account._replace(owner='John')
```

Enumerated constants can be implemented with named tuples, but it is simpler and more efficient to use a simple class declaration:

```
>>> Status = namedtuple('Status', 'open pending closed')._make(range(3))
>>> Status.open, Status.pending, Status.closed
(0, 1, 2)
>>> class Status:
...     open, pending, closed = range(3)
```

See also:

Named tuple recipe adapted for Python 2.4.

8.3.5 OrderedDict objects

Ordered dictionaries are just like regular dictionaries but they remember the order that items were inserted. When iterating over an ordered dictionary, the items are returned in the order their keys were first added.

class collections.OrderedDict([items])
: Return an instance of a dict subclass, supporting the usual dict methods. An OrderedDict is a dict that remembers the order that keys were first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

  New in version 2.7.

OrderedDict.popitem(last=True)
: The popitem() method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if last is true or FIFO order if false.

In addition to the usual mapping methods, ordered dictionaries also support reverse iteration using reversed().

Equality tests between OrderedDict objects are order-sensitive and are implemented as list(od1.items())==list(od2.items()). Equality tests between OrderedDict objects and other Mapping objects are order-insensitive like regular dictionaries. This allows OrderedDict objects to be substituted anywhere a regular dictionary is used.

The OrderedDict constructor and update() method both accept keyword arguments, but their order is lost because Python's function call semantics pass-in keyword arguments using a regular unordered dictionary.

See also:

Equivalent OrderedDict recipe that runs on Python 2.4 or later.

OrderedDict Examples and Recipes

Since an ordered dictionary remembers its insertion order, it can be used in conjunction with sorting to make a sorted dictionary:

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

The new sorted dictionaries maintain their sort order when entries are deleted. But when new keys are added, the keys are appended to the end and the sort is not maintained.

It is also straight-forward to create an ordered dictionary variant that remembers the order the keys were last inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        if key in self:
            del self[key]
        OrderedDict.__setitem__(self, key, value)
```

An ordered dictionary can be combined with the Counter class so that the counter remembers the order elements are first encountered:

```
class OrderedCounter(Counter, OrderedDict):
    'Counter that remembers the order elements are first encountered'

    def __repr__(self):
        return '%s(%r)' % (self.__class__.__name__, OrderedDict(self))

    def __reduce__(self):
        return self.__class__, (OrderedDict(self),)
```

8.3.6 Collections Abstract Base Classes

The collections module offers the following ABCs:

ABC	Inherits from	Abstract Methods	Mixin Methods
Container		__contains__	
Hashable		__hash__	
Iterable		__iter__	
Iterator	Iterable	next	__iter__
Sized		__len__	
Callable		__call__	
Sequence	Sized, Iterable, Container	__getitem__, __len__	__contains__, __iter__, __reversed__, index, and count
MutableSequence	Sequence	__getitem__, __setitem__, __delitem__, __len__, insert	Inherited Sequence methods and append, reverse, extend, pop, remove, and __iadd__
Set	Sized, Iterable, Container	__contains__, __iter__, __len__	__le__, __lt__, __eq__, __ne__, __gt__, __ge__, __and__, __or__, __sub__, __xor__, and isdisjoint
MutableSet	Set	__contains__, __iter__, __len__, add, discard	Inherited Set methods and clear, pop, remove, __ior__, __iand__, __ixor__, and __isub__
Mapping	Sized, Iterable, Container	__getitem__, __iter__, __len__	__contains__, keys, items, values, get, __eq__, and __ne__
MutableMapping	Mapping	__getitem__, __setitem__, __delitem__, __iter__, __len__	Inherited Mapping methods and pop, popitem, clear, update, and setdefault
MappingView	Sized		__len__
ItemsView	MappingView, Set		__contains__, __iter__
KeysView	MappingView, Set		__contains__, __iter__
ValuesView	MappingView		__contains__, __iter__

class collections.Container
class collections.Hashable
class collections.Sized
class collections.Callable
ABCs for classes that provide respectively the methods __contains__(), __hash__(), __len__(), and __call__().

class collections.Iterable
ABC for classes that provide the __iter__() method. See also the definition of iterable.

class collections.Iterator
: ABC for classes that provide the __iter__() and next() methods. See also the definition of iterator.

class collections.Sequence
class collections.MutableSequence
: ABCs for read-only and mutable sequences.

class collections.Set
class collections.MutableSet
: ABCs for read-only and mutable sets.

class collections.Mapping
class collections.MutableMapping
: ABCs for read-only and mutable mappings.

class collections.MappingView
class collections.ItemsView
class collections.KeysView
class collections.ValuesView
: ABCs for mapping, items, keys, and values views.

These ABCs allow us to ask classes or instances if they provide particular functionality, for example:

```
size = None
if isinstance(myvar, collections.Sized):
    size = len(myvar)
```

Several of the ABCs are also useful as mixins that make it easier to develop classes supporting container APIs. For example, to write a class supporting the full Set API, it only necessary to supply the three underlying abstract methods: __contains__(), __iter__(), and __len__(). The ABC supplies the remaining methods such as __and__() and isdisjoint()

```
class ListBasedSet(collections.Set):
     ''' Alternate set implementation favoring space over speed
         and not requiring the set elements to be hashable. '''
     def __init__(self, iterable):
         self.elements = lst = []
         for value in iterable:
             if value not in lst:
                 lst.append(value)

     def __iter__(self):
         return iter(self.elements)

     def __contains__(self, value):
         return value in self.elements

     def __len__(self):
         return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

Notes on using Set and MutableSet as a mixin:

(1) Since some set operations create new sets, the default mixin methods need a way to create new instances from an iterable. The class constructor is assumed to have a signature in the form ClassName(iterable). That assumption is factored-out to an internal classmethod called _from_iterable()

which calls cls(iterable) to produce a new set. If the Set mixin is being used in a class with a different constructor signature, you will need to override _from_iterable() with a classmethod that can construct new instances from an iterable argument.

(2) To override the comparisons (presumably for speed, as the semantics are fixed), redefine __le__() and __ge__(), then the other operations will automatically follow suit.

(3) The Set mixin provides a _hash() method to compute a hash value for the set; however, __hash__() is not defined because not all sets are hashable or immutable. To add set hashability using mixins, inherit from both Set() and Hashable(), then define __hash__ = Set._hash.

See also:

- OrderedSet recipe for an example built on MutableSet.
- For more about ABCs, see the abc module and PEP 3119.

8.4 heapq — Heap queue algorithm

New in version 2.3.

Source code: Lib/heapq.py

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Heaps are binary trees for which every parent node has a value less than or equal to any of its children. This implementation uses arrays for which heap[k] <= heap[2*k+1] and heap[k] <= heap[2*k+2] for all k, counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that its smallest element is always the root, heap[0].

The API below differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Our pop method returns the smallest item, not the largest (called a "min heap" in textbooks; a "max heap" is more common in texts because of its suitability for in-place sorting).

These two make it possible to view the heap as a regular Python list without surprises: heap[0] is the smallest item, and heap.sort() maintains the heap invariant!

To create a heap, use a list initialized to [], or you can transform a populated list into a heap via function heapify().

The following functions are provided:

heapq.heappush(heap, item)
Push the value item onto the heap, maintaining the heap invariant.

heapq.heappop(heap)
Pop and return the smallest item from the heap, maintaining the heap invariant. If the heap is empty, IndexError is raised. To access the smallest item without popping it, use heap[0].

heapq.heappushpop(heap, item)
Push item on the heap, then pop and return the smallest item from the heap. The combined action runs more efficiently than heappush() followed by a separate call to heappop().

New in version 2.6.

heapq.heapify(x)
Transform list x into a heap, in-place, in linear time.

heapq.heapreplace(heap, item)
Pop and return the smallest item from the heap, and also push the new item. The heap size doesn't change. If the heap is empty, IndexError is raised.

This one step operation is more efficient than a heappop() followed by heappush() and can be more appropriate when using a fixed-size heap. The pop/push combination always returns an element from the heap and replaces it with item.

The value returned may be larger than the item added. If that isn't desired, consider using heappushpop() instead. Its push/pop combination returns the smaller of the two values, leaving the larger value on the heap.

The module also offers three general purpose functions based on heaps.

heapq.merge(*iterables)
Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an iterator over the sorted values.

Similar to sorted(itertools.chain(*iterables)) but returns an iterable, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

New in version 2.6.

heapq.nlargest(n, iterable[, key])
Return a list with the n largest elements from the dataset defined by iterable. key, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable: key=str.lower Equivalent to: sorted(iterable, key=key, reverse=True)[:n]

New in version 2.4.

Changed in version 2.5: Added the optional key argument.

heapq.nsmallest(n, iterable[, key])
Return a list with the n smallest elements from the dataset defined by iterable. key, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable: key=str.lower Equivalent to: sorted(iterable, key=key)[:n]

New in version 2.4.

Changed in version 2.5: Added the optional key argument.

The latter two functions perform best for smaller values of n. For larger values, it is more efficient to use the sorted() function. Also, when n==1, it is more efficient to use the built-in min() and max() functions. If repeated usage of these functions is required, consider turning the iterable into an actual heap.

8.4.1 Basic Examples

A heapsort can be implemented by pushing all values onto a heap and then popping off the smallest values one at a time:

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
```

(continues on next page)

(continued from previous page)

```
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This is similar to sorted(iterable), but unlike sorted(), this implementation is not stable.

Heap elements can be tuples. This is useful for assigning comparison values (such as task priorities) alongside the main record being tracked:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.4.2 Priority Queue Implementation Notes

A priority queue is common use for a heap, and it presents several implementation challenges:

- Sort stability: how do you get two tasks with equal priorities to be returned in the order they were originally added?
- In the future with Python 3, tuple comparison breaks for (priority, task) pairs if the priorities are equal and the tasks do not have a default comparison order.
- If the priority of a task changes, how do you move it to a new position in the heap?
- Or if a pending task needs to be deleted, how do you find it and remove it from the queue?

A solution to the first two challenges is to store entries as 3-element list including the priority, an entry count, and the task. The entry count serves as a tie-breaker so that two tasks with the same priority are returned in the order they were added. And since no two entry counts are the same, the tuple comparison will never attempt to directly compare two tasks.

The remaining challenges revolve around finding a pending task and making changes to its priority or removing it entirely. Finding a task can be done with a dictionary pointing to an entry in the queue.

Removing the entry or changing its priority is more difficult because it would break the heap structure invariants. So, a possible solution is to mark the existing entry as removed and add a new entry with the revised priority:

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
```

(continues on next page)

(continued from previous page)

```
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

8.4.3 Theory

Heaps are arrays for which a[k] <= a[2*k+1] and a[k] <= a[2*k+2] for all k, counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that a[0] is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are k, not a[k]:

```
                   0

          1                 2

    3           4           5           6

 7     8     9     10    11    12    13    14

15 16  17 18  19 20  21 22  23 24  25 26  27 28  29 30
```

In the tree above, each cell k is topping 2*k+1 and 2*k+2. In a usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell "wins" over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the "next" winner is to move some loser (let's say cell 30 in the diagram above) into the 0 position, and then percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an O(n log n) sort.

A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not "better" than the last 0'th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the "win" condition means the smallest scheduled time. When an event schedules other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer :-).

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they are reasonably speedy, the speed is almost constant, and the worst case is not much different than

the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing "runs" (which are pre-sorted sequences, whose size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised[1]. It is very important that the initial sort produces the longest runs possible. Tournaments are a good way to achieve that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit the current run, you'll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.

Moreover, if you output the 0'th item on disk and get an input which may not fit in the current tournament (because the value "wins" over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run. Clever and quite effective!

In a word, heaps are useful memory structures to know. I use them in a few applications, and I think it is good to keep a 'heap' module around. :-)

8.5 bisect — Array bisection algorithm

New in version 2.1.

Source code: Lib/bisect.py

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over the more common approach. The module is called bisect because it uses a basic bisection algorithm to do its work. The source code may be most useful as a working example of the algorithm (the boundary conditions are already right!).

The following functions are provided:

bisect.bisect_left(a, x, lo=0, hi=len(a))
: Locate the insertion point for x in a to maintain sorted order. The parameters lo and hi may be used to specify a subset of the list which should be considered; by default the entire list is used. If x is already present in a, the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to list.insert() assuming that a is already sorted.

  The returned insertion point i partitions the array a into two halves so that all(val < x for val in a[lo:i]) for the left side and all(val >= x for val in a[i:hi]) for the right side.

bisect.bisect_right(a, x, lo=0, hi=len(a))
bisect.bisect(a, x, lo=0, hi=len(a))
: Similar to bisect_left(), but returns an insertion point which comes after (to the right of) any existing entries of x in a.

  The returned insertion point i partitions the array a into two halves so that all(val <= x for val in a[lo:i]) for the left side and all(val > x for val in a[i:hi]) for the right side.

[1] The disk balancing algorithms which are current, nowadays, are more annoying than clever, and this is a consequence of the seeking capabilities of the disks. On devices which cannot seek, like big tape drives, the story was quite different, and one had to be very clever to ensure (far in advance) that each tape movement will be the most effective possible (that is, will best participate at "progressing" the merge). Some tapes were even able to read backwards, and this was also used to avoid the rewinding time. Believe me, real good tape sorts were quite spectacular to watch! From all times, sorting has always been a Great Art! :-)

bisect.insort_left(a, x, lo=0, hi=len(a))
: Insert x in a in sorted order. This is equivalent to a.insert(bisect.bisect_left(a, x, lo, hi), x) assuming that a is already sorted. Keep in mind that the O(log n) search is dominated by the slow O(n) insertion step.

bisect.insort_right(a, x, lo=0, hi=len(a))
bisect.insort(a, x, lo=0, hi=len(a))
: Similar to insort_left(), but inserting x in a after any existing entries of x.

See also:

SortedCollection recipe that uses bisect to build a full-featured collection class with straight-forward search methods and support for a key-function. The keys are precomputed to save unnecessary calls to the key function during searches.

8.5.1 Searching Sorted Lists

The above bisect() functions are useful for finding insertion points but can be tricky or awkward to use for common searching tasks. The following five functions show how to transform them into the standard lookups for sorted lists:

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.5.2 Other Examples

The bisect() function can be useful for numeric table lookups. This example uses bisect() to look up a letter grade for an exam score (say) based on a set of ordered numeric breakpoints: 90 and up is an 'A', 80 to 89 is a 'B', and so on:

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
        i = bisect(breakpoints, score)
        return grades[i]

>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

Unlike the sorted() function, it does not make sense for the bisect() functions to have key or reversed arguments because that would lead to an inefficient design (successive calls to bisect functions would not "remember" all of the previous key lookups).

Instead, it is better to search a list of precomputed keys to find the index of the record in question:

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]         # precomputed list of keys
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.6 array — Efficient arrays of numeric values

This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character. The following type codes are defined:

Type code	C Type	Python Type	Minimum size in bytes
'c'	char	character	1
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode character	2 (see note)
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	long	2
'l'	signed long	int	4
'L'	unsigned long	long	4
'f'	float	float	4
'd'	double	float	8

Note: The 'u' typecode corresponds to Python's unicode character. On narrow Unicode builds this is 2-bytes, on wide builds this is 4-bytes.

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the itemsize attribute. The values stored for 'L' and 'I' items will be represented as Python long integers when retrieved, because Python's plain integer type cannot represent the full range of C's unsigned (long) integers.

The module defines the following type:

class array.array(typecode[, initializer])
A new array whose items are restricted by typecode, and initialized from the optional initializer value, which must be a list, string, or iterable over elements of the appropriate type.

Changed in version 2.4: Formerly, only lists or strings were accepted.

If given a list or string, the initializer is passed to the new array's fromlist(), fromstring(), or fromunicode() method (see below) to add initial items to the array. Otherwise, the iterable initializer is passed to the extend() method.

array.ArrayType
Obsolete alias for array.

Array objects support the ordinary sequence operations of indexing, slicing, concatenation, and multiplication. When using slice assignment, the assigned value must be an array object with the same type code; in all other cases, TypeError is raised. Array objects also implement the buffer interface, and may be used wherever buffer objects are supported.

The following data items and methods are also supported:

array.typecode
The typecode character used to create the array.

array.itemsize
The length in bytes of one array item in the internal representation.

array.append(x)
Append a new item with value x to the end of the array.

array.buffer_info()
Return a tuple (address, length) giving the current memory address and the length in elements of the buffer used to hold array's contents. The size of the memory buffer in bytes can be computed as array.buffer_info()[1] * array.itemsize. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain ioctl() operations. The returned numbers are valid as long as the array exists and no length-changing operations are applied to it.

Note: When using array objects from code written in C or C++ (the only way to effectively make use of this information), it makes more sense to use the buffer interface supported by array objects. This method is maintained for backward compatibility and should be avoided in new code. The buffer interface is documented in bufferobjects.

array.byteswap()
"Byteswap" all items of the array. This is only supported for values which are 1, 2, 4, or 8 bytes in size; for other types of values, RuntimeError is raised. It is useful when reading data from a file written on a machine with a different byte order.

array.count(x)
Return the number of occurrences of x in the array.

array.extend(iterable)
Append items from iterable to the end of the array. If iterable is another array, it must have exactly the same type code; if not, TypeError will be raised. If iterable is not an array, it must be iterable and its elements must be the right type to be appended to the array.

Changed in version 2.4: Formerly, the argument could only be another array.

array.fromfile(f, n)
Read n items (as machine values) from the file object f and append them to the end of the array. If less than n items are available, EOFError is raised, but the items that were available are still inserted into the array. f must be a real built-in file object; something else with a read() method won't do.

array.fromlist(list)
Append items from the list. This is equivalent to for x in list: a.append(x) except that if there is a type error, the array is unchanged.

array.fromstring(s)
Appends items from the string, interpreting the string as an array of machine values (as if it had been read from a file using the fromfile() method).

array.fromunicode(s)
Extends this array with data from the given unicode string. The array must be a type 'u' array; otherwise a ValueError is raised. Use array.fromstring(unicodestring.encode(enc)) to append Unicode data to an array of some other type.

array.index(x)
Return the smallest i such that i is the index of the first occurrence of x in the array.

array.insert(i, x)
Insert a new item with value x in the array before position i. Negative values are treated as being relative to the end of the array.

array.pop([i])
Removes the item with the index i from the array and returns it. The optional argument defaults to -1, so that by default the last item is removed and returned.

array.read(f, n)
Deprecated since version 1.5.1: Use the fromfile() method.

Read n items (as machine values) from the file object f and append them to the end of the array. If less than n items are available, EOFError is raised, but the items that were available are still inserted into the array. f must be a real built-in file object; something else with a read() method won't do.

array.remove(x)
Remove the first occurrence of x from the array.

array.reverse()
Reverse the order of the items in the array.

array.tofile(f)
Write all items (as machine values) to the file object f.

array.tolist()
Convert the array to an ordinary list with the same items.

array.tostring()
Convert the array to an array of machine values and return the string representation (the same sequence of bytes that would be written to a file by the tofile() method.)

array.tounicode()
Convert the array to a unicode string. The array must be a type 'u' array; otherwise a ValueError is raised. Use array.tostring().decode(enc) to obtain a unicode string from an array of some other type.

array.write(f)
Deprecated since version 1.5.1: Use the tofile() method.

Write all items (as machine values) to the file object f.

When an array object is printed or converted to a string, it is represented as array(typecode, initializer). The initializer is omitted if the array is empty, otherwise it is a string if the typecode is 'c', otherwise it is a list of numbers. The string is guaranteed to be able to be converted back to an array with the same type and value using eval(), so long as the array class has been imported using from array import array. Examples:

```
array('l')
array('c', 'hello world')
array('u', u'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

See also:

Module struct Packing and unpacking of heterogeneous binary data.

Module xdrlib Packing and unpacking of External Data Representation (XDR) data as used in some remote procedure call systems.

The Numerical Python Documentation The Numeric Python extension (NumPy) defines another array type; see http://www.numpy.org/ for further information about Numerical Python.

8.7 sets — Unordered collections of unique elements

New in version 2.3.

Deprecated since version 2.6: The built-in set/frozenset types replace this module.

The sets module provides classes for constructing and manipulating unordered collections of unique elements. Common uses include membership testing, removing duplicates from a sequence, and computing standard math operations on sets such as intersection, union, difference, and symmetric difference.

Like other collections, sets support x in set, len(set), and for x in set. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

Most set applications use the Set class which provides every set method except for __hash__(). For advanced applications requiring a hash method, the ImmutableSet class adds a __hash__() method but omits methods which alter the contents of the set. Both Set and ImmutableSet derive from BaseSet, an abstract class useful for determining whether something is a set: isinstance(obj, BaseSet).

The set classes are implemented using dictionaries. Accordingly, the requirements for set elements are the same as those for dictionary keys; namely, that the element defines both __eq__() and __hash__(). As a result, sets cannot contain mutable elements such as lists or dictionaries. However, they can contain immutable collections such as tuples or instances of ImmutableSet. For convenience in implementing sets of sets, inner sets are automatically converted to immutable form, for example, Set([Set(['dog'])]) is transformed to Set([ImmutableSet(['dog'])]).

class sets.Set([iterable])
Constructs a new empty Set object. If the optional iterable parameter is supplied, updates the set

with elements obtained from iteration. All of the elements in iterable should be immutable or be transformable to an immutable using the protocol described in section Protocol for automatic conversion to immutable.

class sets.ImmutableSet([iterable])
Constructs a new empty ImmutableSet object. If the optional iterable parameter is supplied, updates the set with elements obtained from iteration. All of the elements in iterable should be immutable or be transformable to an immutable using the protocol described in section Protocol for automatic conversion to immutable.

Because ImmutableSet objects provide a __hash__() method, they can be used as set elements or as dictionary keys. ImmutableSet objects do not have methods for adding or removing elements, so all of the elements must be known when the constructor is called.

8.7.1 Set Objects

Instances of Set and ImmutableSet both provide the following operations:

Operation	Equivalent	Result
len(s)		number of elements in set s (cardinality)
x in s		test x for membership in s
x not in s		test x for non-membership in s
s.issubset(t)	s <= t	test whether every element in s is in t
s.issuperset(t)	s >= t	test whether every element in t is in s
s.union(t)	s | t	new set with elements from both s and t
s.intersection(t)	s & t	new set with elements common to s and t
s.difference(t)	s - t	new set with elements in s but not in t
s.symmetric_difference(t)	s ^ t	new set with elements in either s or t but not both
s.copy()		new set with a shallow copy of s

Note, the non-operator versions of union(), intersection(), difference(), and symmetric_difference() will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like Set('abc') & 'cbs' in favor of the more readable Set('abc').intersection('cbs').

Changed in version 2.3.1: Formerly all arguments were required to be sets.

In addition, both Set and ImmutableSet support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

The subset and equality comparisons do not generalize to a complete ordering function. For example, any two disjoint sets are not equal and are not subsets of each other, so all of the following return False: a<b, a==b, or a>b. Accordingly, sets do not implement the __cmp__() method.

Since sets only define partial ordering (subset relationships), the output of the list.sort() method is undefined for lists of sets.

The following table lists operations available in ImmutableSet but not found in Set:

Operation	Result
hash(s)	returns a hash value for s

The following table lists operations available in Set but not found in ImmutableSet:

Operation	Equivalent	Result
s.update(t)	s |= t	return set s with elements added from t
s.intersection_update(t)	s &= t	return set s keeping only elements also found in t
s.difference_update(t)	s -= t	return set s after removing elements found in t
s.symmetric_difference_update(t)	s ^= t	return set s with elements from s or t but not both
s.add(x)		add element x to set s
s.remove(x)		remove x from set s; raises KeyError if not present
s.discard(x)		removes x from set s if present
s.pop()		remove and return an arbitrary element from s; raises KeyError if empty
s.clear()		remove all elements from set s

Note, the non-operator versions of update(), intersection_update(), difference_update(), and symmetric_difference_update() will accept any iterable as an argument.

Changed in version 2.3.1: Formerly all arguments were required to be sets.

Also note, the module also includes a union_update() method which is an alias for update(). The method is included for backwards compatibility. Programmers should prefer the update() method because it is supported by the built-in set() and frozenset() types.

8.7.2 Example

```
>>> from sets import Set
>>> engineers = Set(['John', 'Jane', 'Jack', 'Janice'])
>>> programmers = Set(['Jack', 'Sam', 'Susan', 'Janice'])
>>> managers = Set(['Jane', 'Jack', 'Susan', 'Zack'])
>>> employees = engineers | programmers | managers          # union
>>> engineering_management = engineers & managers           # intersection
>>> fulltime_management = managers - engineers - programmers # difference
>>> engineers.add('Marvin')                                 # add element
>>> print engineers # doctest: +SKIP
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
>>> employees.issuperset(engineers)     # superset test
False
>>> employees.update(engineers)         # update from another set
>>> employees.issuperset(engineers)
True
>>> for group in [engineers, programmers, managers, employees]: # doctest: +SKIP
...     group.discard('Susan')          # unconditionally remove element
...     print group
...
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
Set(['Janice', 'Jack', 'Sam'])
Set(['Jane', 'Zack', 'Jack'])
Set(['Jack', 'Sam', 'Jane', 'Marvin', 'Janice', 'John', 'Zack'])
```

8.7.3 Protocol for automatic conversion to immutable

Sets can only contain immutable elements. For convenience, mutable Set objects are automatically copied to an ImmutableSet before being added as a set element.

The mechanism is to always add a hashable element, or if it is not hashable, the element is checked to see if it has an __as_immutable__() method which returns an immutable equivalent.

Since Set objects have a __as_immutable__() method returning an instance of ImmutableSet, it is possible to construct sets of sets.

A similar mechanism is needed by the __contains__() and remove() methods which need to hash an element to check for membership in a set. Those methods check an element for hashability and, if not, check for a __as_temporarily_immutable__() method which returns the element wrapped by a class that provides temporary methods for __hash__(), __eq__(), and __ne__().

The alternate mechanism spares the need to build a separate copy of the original mutable object.

Set objects implement the __as_temporarily_immutable__() method which returns the Set object wrapped by a new class _TemporarilyImmutableSet.

The two mechanisms for adding hashability are normally invisible to the user; however, a conflict can arise in a multi-threaded environment where one thread is updating a set while another has temporarily wrapped it in _TemporarilyImmutableSet. In other words, sets of mutable sets are not thread-safe.

8.7.4 Comparison to the built-in set types

The built-in set and frozenset types were designed based on lessons learned from the sets module. The key differences are:

- Set and ImmutableSet were renamed to set and frozenset.
- There is no equivalent to BaseSet. Instead, use isinstance(x, (set, frozenset)).
- The hash algorithm for the built-ins performs significantly better (fewer collisions) for most datasets.
- The built-in versions have more space efficient pickles.
- The built-in versions do not have a union_update() method. Instead, use the update() method which is equivalent.
- The built-in versions do not have a _repr(sorted=True) method. Instead, use the built-in repr() and sorted() functions: repr(sorted(s)).
- The built-in version does not have a protocol for automatic conversion to immutable. Many found this feature to be confusing and no one in the community reported having found real uses for it.

8.8 sched — Event scheduler

Source code: Lib/sched.py

The sched module defines a class which implements a general purpose event scheduler:

class sched.scheduler(timefunc, delayfunc)
: The scheduler class defines a generic interface to scheduling events. It needs two functions to actually deal with the "outside world" — timefunc should be callable without arguments, and return a number (the "time", in any units whatsoever). The delayfunc function should be callable with one argument,

compatible with the output of timefunc, and should delay that many time units. delayfunc will also be called with the argument 0 after each event is run to allow other threads an opportunity to run in multi-threaded applications.

Example:

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(): print "From print_time", time.time()
...
>>> def print_some_times():
...     print time.time()
...     s.enter(5, 1, print_time, ())
...     s.enter(10, 1, print_time, ())
...     s.run()
...     print time.time()
...
>>> print_some_times()
930343690.257
From print_time 930343695.274
From print_time 930343700.273
930343700.276
```

In multi-threaded environments, the scheduler class has limitations with respect to thread-safety, inability to insert a new task before the one currently pending in a running scheduler, and holding up the main thread until the event queue is empty. Instead, the preferred approach is to use the threading.Timer class instead.

Example:

```
>>> import time
>>> from threading import Timer
>>> def print_time():
...     print "From print_time", time.time()
...
>>> def print_some_times():
...     print time.time()
...     Timer(5, print_time, ()).start()
...     Timer(10, print_time, ()).start()
...     time.sleep(11)  # sleep while time-delay events execute
...     print time.time()
...
>>> print_some_times()
930343690.257
From print_time 930343695.274
From print_time 930343700.273
930343701.301
```

8.8.1 Scheduler Objects

scheduler instances have the following methods and attributes:

scheduler.enterabs(time, priority, action, argument)
: Schedule a new event. The time argument should be a numeric type compatible with the return value of the timefunc function passed to the constructor. Events scheduled for the same time will be executed in the order of their priority. A lower number represents a higher priority.

  Executing the event means executing action(*argument). argument must be a sequence holding the parameters for action.

Return value is an event which may be used for later cancellation of the event (see cancel()).

scheduler.enter(delay, priority, action, argument)
Schedule an event for delay more time units. Other than the relative time, the other arguments, the effect and the return value are the same as those for enterabs().

scheduler.cancel(event)
Remove the event from the queue. If event is not an event currently in the queue, this method will raise a ValueError.

scheduler.empty()
Return true if the event queue is empty.

scheduler.run()
Run all scheduled events. This function will wait (using the delayfunc() function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.

Either action or delayfunc can raise an exception. In either case, the scheduler will maintain a consistent state and propagate the exception. If an exception is raised by action, the event will not be attempted in future calls to run().

If a sequence of events takes longer to run than the time available before the next event, the scheduler will simply fall behind. No events will be dropped; the calling code is responsible for canceling events which are no longer pertinent.

scheduler.queue
Read-only attribute returning a list of upcoming events in the order they will be run. Each event is shown as a named tuple with the following fields: time, priority, action, argument.

New in version 2.6.

8.9 mutex — Mutual exclusion support

Deprecated since version 2.6: The mutex module has been removed in Python 3.

The mutex module defines a class that allows mutual-exclusion via acquiring and releasing locks. It does not require (or imply) threading or multi-tasking, though it could be useful for those purposes.

The mutex module defines the following class:

class mutex.mutex
Create a new (unlocked) mutex.

A mutex has two pieces of state — a "locked" bit and a queue. When the mutex is not locked, the queue is empty. Otherwise, the queue contains zero or more (function, argument) pairs representing functions (or methods) waiting to acquire the lock. When the mutex is unlocked while the queue is not empty, the first queue entry is removed and its function(argument) pair called, implying it now has the lock.

Of course, no multi-threading is implied – hence the funny interface for lock(), where a function is called once the lock is acquired.

8.9.1 Mutex Objects

mutex objects have following methods:

mutex.test()
Check whether the mutex is locked.

mutex.testandset()
"Atomic" test-and-set, grab the lock if it is not set, and return True, otherwise, return False.

mutex.lock(function, argument)
Execute function(argument), unless the mutex is locked. In the case it is locked, place the function and argument on the queue. See unlock() for explanation of when function(argument) is executed in that case.

mutex.unlock()
Unlock the mutex if queue is empty, otherwise execute the first element in the queue.

8.10 Queue — A synchronized queue class

Note: The Queue module has been renamed to queue in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/Queue.py

The Queue module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The Queue class in this module implements all the required locking semantics. It depends on the availability of thread support in Python; see the threading module.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the heapq module) and the lowest valued entry is retrieved first.

The Queue module defines the following classes and exceptions:

class Queue.Queue(maxsize=0)
Constructor for a FIFO queue. maxsize is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If maxsize is less than or equal to zero, the queue size is infinite.

class Queue.LifoQueue(maxsize=0)
Constructor for a LIFO queue. maxsize is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If maxsize is less than or equal to zero, the queue size is infinite.

New in version 2.6.

class Queue.PriorityQueue(maxsize=0)
Constructor for a priority queue. maxsize is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If maxsize is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one returned by sorted(list(entries))[0]). A typical pattern for entries is a tuple in the form: (priority_number, data).

New in version 2.6.

exception Queue.Empty
Exception raised when non-blocking get() (or get_nowait()) is called on a Queue object which is empty.

exception Queue.Full
Exception raised when non-blocking put() (or put_nowait()) is called on a Queue object which is full.

See also:

collections.deque is an alternative implementation of unbounded queues with fast atomic append() and popleft() operations that do not require locking.

8.10.1 Queue Objects

Queue objects (Queue, LifoQueue, or PriorityQueue) provide the public methods described below.

Queue.qsize()
Return the approximate size of the queue. Note, qsize() > 0 doesn't guarantee that a subsequent get() will not block, nor will qsize() < maxsize guarantee that put() will not block.

Queue.empty()
Return True if the queue is empty, False otherwise. If empty() returns True it doesn't guarantee that a subsequent call to put() will not block. Similarly, if empty() returns False it doesn't guarantee that a subsequent call to get() will not block.

Queue.full()
Return True if the queue is full, False otherwise. If full() returns True it doesn't guarantee that a subsequent call to get() will not block. Similarly, if full() returns False it doesn't guarantee that a subsequent call to put() will not block.

Queue.put(item[, block[, timeout]])
Put item into the queue. If optional args block is true and timeout is None (the default), block if necessary until a free slot is available. If timeout is a positive number, it blocks at most timeout seconds and raises the Full exception if no free slot was available within that time. Otherwise (block is false), put an item on the queue if a free slot is immediately available, else raise the Full exception (timeout is ignored in that case).

New in version 2.3: The timeout parameter.

Queue.put_nowait(item)
Equivalent to put(item, False).

Queue.get([block[, timeout]])
Remove and return an item from the queue. If optional args block is true and timeout is None (the default), block if necessary until an item is available. If timeout is a positive number, it blocks at most timeout seconds and raises the Empty exception if no item was available within that time. Otherwise (block is false), return an item if one is immediately available, else raise the Empty exception (timeout is ignored in that case).

New in version 2.3: The timeout parameter.

Queue.get_nowait()
Equivalent to get(False).

Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemon consumer threads.

Queue.task_done()
Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

New in version 2.5.

Queue.join()
Blocks until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, join() unblocks.

New in version 2.5.

Example of how to wait for enqueued tasks to be completed:

```
def worker():
    while True:
        item = q.get()
        do_work(item)
        q.task_done()

q = Queue()
for i in range(num_worker_threads):
     t = Thread(target=worker)
     t.daemon = True
     t.start()

for item in source():
    q.put(item)

q.join()       # block until all tasks are done
```

8.11 weakref — Weak references

New in version 2.1.

Source code: Lib/weakref.py

The weakref module allows the Python programmer to create weak references to objects.

In the following, the term referent means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, garbage collection is free to destroy the referent and reuse its memory for something else. A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The WeakKeyDictionary and WeakValueDictionary classes supplied by the weakref module are an alternative, using weak references to construct mappings that don't keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a WeakValueDictionary, then when the last remaining references to that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.

WeakKeyDictionary and WeakValueDictionary use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. Most programs should find that using one of these weak dictionary types is all they need – it's not usually necessary to create your own weak references directly. The low-level machinery used by the weak dictionary implementations is exposed by the weakref module for the benefit of advanced uses.

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), methods (both bound and unbound), sets, frozensets, file objects, generators, type objects, DBcursor objects from the bsddb module, sockets, arrays, deques, regular expression pattern objects, and code objects.

Changed in version 2.4: Added support for files, sockets, arrays, and patterns.

Changed in version 2.7: Added support for thread.lock, threading.Lock, and code objects.

Several built-in types such as list and dict do not directly support weak references but can add support through subclassing:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

CPython implementation detail: Other built-in types such as tuple and long do not support weak references even when subclassed.

Extension types can easily be made to support weak references; see weakref-support.

class weakref.ref(object[, callback])
: Return a weak reference to object. The original object can be retrieved by calling the reference object if the referent is still alive; if the referent is no longer alive, calling the reference object will cause None to be returned. If callback is provided and not None, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no longer be available.

  It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

  Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are handled in exactly the same way as exceptions raised from an object's __del__() method.

  Weak references are hashable if the object is hashable. They will maintain their hash value even after the object was deleted. If hash() is called the first time only after the object was deleted, the call will raise TypeError.

  Weak references support tests for equality, but not ordering. If the referents are still alive, two references have the same equality relationship as their referents (regardless of the callback). If either referent has been deleted, the references are equal only if the reference objects are the same object.

  Changed in version 2.4: This is now a subclassable type rather than a factory function; it derives from object.

weakref.proxy(object[, callback])
: Return a proxy to object which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either ProxyType or CallableProxyType, depending on whether object is callable. Proxy objects are not hashable regardless of the referent; this avoids a number of problems related to their fundamentally mutable nature, and prevent their use as dictionary keys. callback is the same as the parameter of the same name to the ref() function.

weakref.getweakrefcount(object)
Return the number of weak references and proxies which refer to object.

weakref.getweakrefs(object)
Return a list of all weak reference and proxy objects which refer to object.

class weakref.WeakKeyDictionary([dict])
Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

Note: Caution: Because a WeakKeyDictionary is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a WeakKeyDictionary because actions performed by the program during iteration may cause items in the dictionary to vanish "by magic" (as a side effect of garbage collection).

WeakKeyDictionary objects have the following additional methods. These expose the internal references directly. The references are not guaranteed to be "live" at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

WeakKeyDictionary.iterkeyrefs()
Return an iterable of the weak references to the keys.

New in version 2.5.

WeakKeyDictionary.keyrefs()
Return a list of weak references to the keys.

New in version 2.5.

class weakref.WeakValueDictionary([dict])
Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

Note: Caution: Because a WeakValueDictionary is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a WeakValueDictionary because actions performed by the program during iteration may cause items in the dictionary to vanish "by magic" (as a side effect of garbage collection).

WeakValueDictionary objects have the following additional methods. These methods have the same issues as the iterkeyrefs() and keyrefs() methods of WeakKeyDictionary objects.

WeakValueDictionary.itervaluerefs()
Return an iterable of the weak references to the values.

New in version 2.5.

WeakValueDictionary.valuerefs()
Return a list of weak references to the values.

New in version 2.5.

class weakref.WeakSet([elements])
Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.

New in version 2.7.

weakref.ReferenceType
The type object for weak references objects.

weakref.ProxyType
The type object for proxies of objects which are not callable.

weakref.CallableProxyType
The type object for proxies of callable objects.

weakref.ProxyTypes
Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

exception weakref.ReferenceError
Exception raised when a proxy object is used but the underlying object has been collected. This is the same as the standard ReferenceError exception.

See also:

PEP 205 - Weak References The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.

8.11.1 Weak Reference Objects

Weak reference objects have no attributes or methods, but do allow the referent to be obtained, if it still exists, by calling it:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns None:

```
>>> del o, o2
>>> print r()
None
```

Testing that a weak reference object is still live should be done using the expression ref() is not None. Normally, application code that needs to use a reference object should follow this pattern:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print "Object has been deallocated; can't frobnicate."
else:
    print "Object is still live!"
    o.do_something_useful()
```

Using a separate test for "liveness" creates race conditions in threaded applications; another thread can cause a weak reference to become invalidated before the weak reference is called; the idiom shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of ref objects can be created through subclassing. This is used in the implementation of the WeakValueDictionary to reduce the memory overhead for each entry in the mapping. This may be most useful to associate additional information with a reference, but could also be used to insert additional processing on calls to retrieve the referent.

This example shows how a subclass of ref can be used to store additional information about an object and affect the value that's returned when the referent is accessed:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.iteritems():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super(ExtendedRef, self).__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.11.2 Example

This simple example shows how an application can use object IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.12 UserDict — Class wrapper for dictionary objects

Source code: Lib/UserDict.py

The module defines a mixin, DictMixin, defining all dictionary methods for classes that already have a minimum mapping interface. This greatly simplifies writing classes that need to be substitutable for dictionaries (such as the shelve module).

This module also defines a class, UserDict, that acts as a wrapper around dictionary objects. The need for this class has been largely supplanted by the ability to subclass directly from dict (a feature that became available starting with Python version 2.2). Prior to the introduction of dict, the UserDict class was used to create dictionary-like sub-classes that obtained new behaviors by overriding existing methods or adding new ones.

The UserDict module defines the UserDict class and DictMixin:

class UserDict.UserDict([initialdata])
Class that simulates a dictionary. The instance's contents are kept in a regular dictionary, which is accessible via the data attribute of UserDict instances. If initialdata is provided, data is initialized with its contents; note that a reference to initialdata will not be kept, allowing it be used for other purposes.

Note: For backward compatibility, instances of UserDict are not iterable.

class UserDict.IterableUserDict([initialdata])
Subclass of UserDict that supports direct iteration (e.g. for key in myDict).

In addition to supporting the methods and operations of mappings (see section Mapping Types — dict), UserDict and IterableUserDict instances provide the following attribute:

IterableUserDict.data
A real dictionary used to store the contents of the UserDict class.

class UserDict.DictMixin
Mixin defining all dictionary methods for classes that already have a minimum dictionary interface including __getitem__(), __setitem__(), __delitem__(), and keys().

This mixin should be used as a superclass. Adding each of the above methods adds progressively more functionality. For instance, defining all but __delitem__() will preclude only pop() and popitem() from the full interface.

In addition to the four base methods, progressively more efficiency comes with defining __contains__(), __iter__(), and iteritems().

Since the mixin has no knowledge of the subclass constructor, it does not define __init__() or copy().

Starting with Python version 2.6, it is recommended to use collections.MutableMapping instead of DictMixin.

Note that DictMixin does not implement the viewkeys(), viewvalues(), or viewitems() methods.

8.13 UserList — Class wrapper for list objects

Note: When Python 2.2 was released, many of the use cases for this class were subsumed by the ability to subclass list directly. However, a handful of use cases remain.

This module provides a list-interface around an underlying data store. By default, that data store is a list; however, it can be used to wrap a list-like interface around other objects (such as persistent storage).

In addition, this class can be mixed-in with built-in classes using multiple inheritance. This can sometimes be useful. For example, you can inherit from UserList and str at the same time. That would not be possible with both a real list and a real str.

This module defines a class that acts as a wrapper around list objects. It is a useful base class for your own list-like classes, which can inherit from them and override existing methods or add new ones. In this way one can add new behaviors to lists.

The UserList module defines the UserList class:

class UserList.UserList([list])
: Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via the data attribute of UserList instances. The instance's contents are initially set to a copy of list, defaulting to the empty list []. list can be any iterable, e.g. a real Python list or a UserList object.

  Note: The UserList class has been moved to the collections module in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

In addition to supporting the methods and operations of mutable sequences (see section Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange), UserList instances provide the following attribute:

UserList.data
: A real Python list object used to store the contents of the UserList class.

Subclassing requirements: Subclasses of UserList are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

Changed in version 2.0: Python versions 1.5.2 and 1.6 also required that the constructor be callable with no parameters, and offer a mutable data attribute. Earlier versions of Python did not attempt to create instances of the derived class.

8.14 UserString — Class wrapper for string objects

Note: This UserString class from this module is available for backward compatibility only. If you are writing code that does not need to work with versions of Python earlier than Python 2.2, please consider subclassing directly from the built-in str type instead of using UserString (there is no built-in equivalent to MutableString).

This module defines a class that acts as a wrapper around string objects. It is a useful base class for your own string-like classes, which can inherit from them and override existing methods or add new ones. In this way one can add new behaviors to strings.

It should be noted that these classes are highly inefficient compared to real string or Unicode objects; this is especially the case for MutableString.

The UserString module defines the following classes:

class UserString.UserString([sequence])
Class that simulates a string or a Unicode string object. The instance's content is kept in a regular string or Unicode string object, which is accessible via the data attribute of UserString instances. The instance's contents are initially set to a copy of sequence. sequence can be either a regular Python string or Unicode string, an instance of UserString (or a subclass) or an arbitrary sequence which can be converted into a string using the built-in str() function.

Note: The UserString class has been moved to the collections module in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

---

class UserString.MutableString([sequence])
This class is derived from the UserString above and redefines strings to be mutable. Mutable strings can't be used as dictionary keys, because dictionaries require immutable objects as keys. The main intention of this class is to serve as an educational example for inheritance and necessity to remove (override) the __hash__() method in order to trap attempts to use a mutable object as dictionary key, which would be otherwise very error prone and hard to track down.

Deprecated since version 2.6: The MutableString class has been removed in Python 3.

In addition to supporting the methods and operations of string and Unicode objects (see section String Methods), UserString instances provide the following attribute:

MutableString.data
A real Python string or Unicode object used to store the content of the UserString class.

8.15 types — Names for built-in types

Source code: Lib/types.py

This module defines names for some object types that are used by the standard Python interpreter, but not for the types defined by various extension modules. Also, it does not include some of the types that arise during processing such as the listiterator type. It is safe to use from types import * — the module does not export any names besides the ones listed here. New names exported by future versions of this module will all end in Type.

Typical use is for functions that do different things depending on their argument types, like the following:

```
from types import *
def delete(mylist, item):
    if type(item) is IntType:
        del mylist[item]
    else:
        mylist.remove(item)
```

Starting in Python 2.2, built-in factory functions such as int() and str() are also names for the corresponding types. This is now the preferred way to access the type instead of using the types module. Accordingly, the example above should be written as follows:

```
def delete(mylist, item):
    if isinstance(item, int):
        del mylist[item]
    else:
        mylist.remove(item)
```

The module defines the following names:

types.NoneType
: The type of None.

types.TypeType
: The type of type objects (such as returned by type()); alias of the built-in type.

types.BooleanType
: The type of the bool values True and False; alias of the built-in bool.

  New in version 2.3.

types.IntType
: The type of integers (e.g. 1); alias of the built-in int.

types.LongType
: The type of long integers (e.g. 1L); alias of the built-in long.

types.FloatType
: The type of floating point numbers (e.g. 1.0); alias of the built-in float.

types.ComplexType
: The type of complex numbers (e.g. 1.0j). This is not defined if Python was built without complex number support.

types.StringType
: The type of character strings (e.g. 'Spam'); alias of the built-in str.

types.UnicodeType
: The type of Unicode character strings (e.g. u'Spam'). This is not defined if Python was built without Unicode support. It's an alias of the built-in unicode.

types.TupleType
: The type of tuples (e.g. (1, 2, 3, 'Spam')); alias of the built-in tuple.

types.ListType
: The type of lists (e.g. [0, 1, 2, 3]); alias of the built-in list.

types.DictType
: The type of dictionaries (e.g. {'Bacon': 1, 'Ham': 0}); alias of the built-in dict.

types.DictionaryType
: An alternate name for DictType.

types.FunctionType
types.LambdaType
: The type of user-defined functions and functions created by lambda expressions.

types.GeneratorType
: The type of generator-iterator objects, produced by calling a generator function.

  New in version 2.2.

types.CodeType
: The type for code objects such as returned by compile().

types.ClassType
: The type of user-defined old-style classes.

types.InstanceType
: The type of instances of user-defined old-style classes.

types.MethodType
: The type of methods of user-defined class instances.

types.UnboundMethodType
: An alternate name for MethodType.

types.BuiltinFunctionType
types.BuiltinMethodType
: The type of built-in functions like len() or sys.exit(), and methods of built-in classes. (Here, the term "built-in" means "written in C".)

types.ModuleType
: The type of modules.

types.FileType
: The type of open file objects such as sys.stdout; alias of the built-in file.

types.XRangeType
: The type of range objects returned by xrange(); alias of the built-in xrange.

types.SliceType
: The type of objects returned by slice(); alias of the built-in slice.

types.EllipsisType
: The type of Ellipsis.

types.TracebackType
: The type of traceback objects such as found in sys.exc_traceback.

types.FrameType
: The type of frame objects such as found in tb.tb_frame if tb is a traceback object.

types.BufferType
: The type of buffer objects created by the buffer() function.

types.DictProxyType
: The type of dict proxies, such as TypeType.__dict__.

types.NotImplementedType
: The type of NotImplemented

types.GetSetDescriptorType
: The type of objects defined in extension modules with PyGetSetDef, such as FrameType.f_locals or array.array.typecode. This type is used as descriptor for object attributes; it has the same purpose as the property type, but for classes defined in extension modules.

  New in version 2.5.

types.MemberDescriptorType
: The type of objects defined in extension modules with PyMemberDef, such as datetime.timedelta.days. This type is used as descriptor for simple C data members which use standard conversion functions; it has the same purpose as the property type, but for classes defined in extension modules.

  CPython implementation detail: In other implementations of Python, this type may be identical to GetSetDescriptorType.

  New in version 2.5.

types.StringTypes
: A sequence containing StringType and UnicodeType used to facilitate easier checking for any string object. Using this is more portable than using a sequence of the two string types constructed elsewhere since it only contains UnicodeType if it has been built in the running version of Python. For example: isinstance(s, types.StringTypes).

New in version 2.2.

8.16 new — Creation of runtime internal objects

Deprecated since version 2.6: The new module has been removed in Python 3. Use the types module's classes instead.

The new module allows an interface to the interpreter object creation functions. This is for use primarily in marshal-type functions, when a new object needs to be created "magically" and not by using the regular creation functions. This module provides a low-level interface to the interpreter, so care must be exercised when using this module. It is possible to supply non-sensical arguments which crash the interpreter when the object is used.

The new module defines the following functions:

new.instance(class[, dict])
: This function creates an instance of class with dictionary dict without calling the __init__() constructor. If dict is omitted or None, a new, empty dictionary is created for the new instance. Note that there are no guarantees that the object will be in a consistent state.

new.instancemethod(function, instance, class)
: This function will return a method object, bound to instance, or unbound if instance is None. function must be callable.

new.function(code, globals[, name[, argdefs[, closure]]])
: Returns a (Python) function with the given code and globals. If name is given, it must be a string or None. If it is a string, the function will have the given name, otherwise the function name will be taken from code.co_name. If argdefs is given, it must be a tuple and will be used to determine the default values of parameters. If closure is given, it must be None or a tuple of cell objects containing objects to bind to the names in code.co_freevars.

new.code(argcount, nlocals, stacksize, flags, codestring, constants, names, varnames, filename, name, firstlineno, lnotab)
: This function is an interface to the PyCode_New() C function.

new.module(name[, doc])
: This function returns a new module object with name name. name must be a string. The optional doc argument can have any type.

new.classobj(name, baseclasses, dict)
: This function returns a new class object, with name name, derived from baseclasses (which should be a tuple of classes) and with namespace dict.

8.17 copy — Shallow and deep copy operations

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other. This module provides generic shallow and deep copy operations (explained below).

Interface summary:

copy.copy(x)
: Return a shallow copy of x.

copy.deepcopy(x)
Return a deep copy of x.

exception copy.error
Raised for module specific errors.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

Two problems often exist with deep copy operations that don't exist with shallow copy operations:

- Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.
- Because deep copy copies everything it may copy too much, such as data which is intended to be shared between copies.

The deepcopy() function avoids these problems by:

- keeping a "memo" dictionary of objects already copied during the current copying pass; and
- letting user-defined classes override the copying operation or the set of components copied.

This module does not copy types like module, method, stack trace, stack frame, file, socket, window, array, or any similar types. It does "copy" functions and classes (shallow and deeply), by returning the original object unchanged; this is compatible with the way these are treated by the pickle module.

Shallow copies of dictionaries can be made using dict.copy(), and of lists by assigning a slice of the entire list, for example, copied_list = original_list[:].

Changed in version 2.5: Added copying functions.

Classes can use the same interfaces to control copying that they use to control pickling. See the description of module pickle for information on these methods. The copy module does not use the copy_reg registration module.

In order for a class to define its own copy implementation, it can define special methods __copy__() and __deepcopy__(). The former is called to implement the shallow copy operation; no additional arguments are passed. The latter is called to implement the deep copy operation; it is passed one argument, the memo dictionary. If the __deepcopy__() implementation needs to make a deep copy of a component, it should call the deepcopy() function with the component as first argument and the memo dictionary as second argument.

See also:

Module pickle Discussion of the special methods used to support object state retrieval and restoration.

8.18 pprint — Data pretty printer

Source code: Lib/pprint.py

The pprint module provides a capability to "pretty-print" arbitrary Python data structures in a form which can be used as input to the interpreter. If the formatted structures include objects which are not fundamental Python types, the representation may not be loadable. This may be the case if objects such as files, sockets,

classes, or instances are included, as well as many other built-in objects which are not representable as Python constants.

The formatted representation keeps objects on a single line if it can, and breaks them onto multiple lines if they don't fit within the allowed width. Construct PrettyPrinter objects explicitly if you need to adjust the width constraint.

Changed in version 2.5: Dictionaries are sorted by key before the display is computed; before 2.5, a dictionary was sorted only if its display required more than one line, although that wasn't documented.

Changed in version 2.6: Added support for set and frozenset.

The pprint module defines one class:

class pprint.PrettyPrinter(indent=1, width=80, depth=None, stream=None)
Construct a PrettyPrinter instance. This constructor understands several keyword parameters. An output stream may be set using the stream keyword; the only method used on the stream object is the file protocol's write() method. If not specified, the PrettyPrinter adopts sys.stdout. Three additional parameters may be used to control the formatted representation. The keywords are indent, depth, and width. The amount of indentation added for each recursive level is specified by indent; the default is one. Other values can cause output to look a little odd, but can make nesting easier to spot. The number of levels which may be printed is controlled by depth; if the data structure being printed is too deep, the next contained level is replaced by .... By default, there is no constraint on the depth of the objects being formatted. The desired output width is constrained using the width parameter; the default is 80 characters. If a structure cannot be formatted within the constrained width, a best effort will be made.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

The PrettyPrinter class supports several derivative functions:

pprint.pformat(object, indent=1, width=80, depth=None)
Return the formatted representation of object as a string. indent, width and depth will be passed to the PrettyPrinter constructor as formatting parameters.

Changed in version 2.4: The parameters indent, width and depth were added.

pprint.pprint(object, stream=None, indent=1, width=80, depth=None)
Prints the formatted representation of object on stream, followed by a newline. If stream is None, sys.stdout is used. This may be used in the interactive interpreter instead of a print statement for inspecting values. indent, width and depth will be passed to the PrettyPrinter constructor as formatting parameters.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

Changed in version 2.4: The parameters indent, width and depth were added.

pprint.isreadable(object)
Determine if the formatted representation of object is "readable," or can be used to reconstruct the value using eval(). This always returns False for recursive objects.

```
>>> pprint.isreadable(stuff)
False
```

pprint.isrecursive(object)
Determine if object requires a recursive representation.

One more support function is also defined:

pprint.saferepr(object)
Return a string representation of object, protected against recursive data structures. If the representation of object exposes a recursive entry, the recursive reference will be represented as <Recursion on typename with id=number>. The representation is not otherwise formatted.

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

8.18.1 PrettyPrinter Objects

PrettyPrinter instances have the following methods:

PrettyPrinter.pformat(object)
Return the formatted representation of object. This takes into account the options passed to the PrettyPrinter constructor.

PrettyPrinter.pprint(object)
Print the formatted representation of object on the configured stream, followed by a newline.

The following methods provide the implementations for the corresponding functions of the same names. Using these methods on an instance is slightly more efficient since new PrettyPrinter objects don't need to be created.

PrettyPrinter.isreadable(object)
Determine if the formatted representation of the object is "readable," or can be used to reconstruct the value using eval(). Note that this returns False for recursive objects. If the depth parameter of the PrettyPrinter is set and the object is deeper than allowed, this returns False.

PrettyPrinter.isrecursive(object)
Determine if the object requires a recursive representation.

This method is provided as a hook to allow subclasses to modify the way objects are converted to strings. The default implementation uses the internals of the saferepr() implementation.

PrettyPrinter.format(object, context, maxlevels, level)

Returns three values: the formatted version of object as a string, a flag indicating whether the result is readable, and a flag indicating whether recursion was detected. The first argument is the object to be presented. The second is a dictionary which contains the id() of objects that are part of the current presentation context (direct and indirect containers for object that are affecting the presentation) as the keys; if an object needs to be presented which is already represented in context, the third return value should be True. Recursive calls to the format() method should add additional entries for containers to this dictionary. The third argument, maxlevels, gives the requested limit to recursion; this will be 0 if there is no requested limit. This argument should be passed unmodified to recursive calls. The fourth argument, level, gives the current level; recursive calls should be passed a value less than that of the current call.

New in version 2.3.

8.18.2 pprint Example

This example demonstrates several uses of the pprint() function and its parameters.

```
>>> import pprint
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))
>>> stuff = ['a' * 10, tup, ['a' * 30, 'b' * 30], ['c' * 20, 'd' * 20]]
>>> pprint.pprint(stuff)
['aaaaaaaaaa',
 ('spam',
  ('eggs',
   ('lumberjack',
    ('knights', ('ni', ('dead', ('parrot', ('fresh fruit',)))))))),
 ['aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa', 'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'],
 ['cccccccccccccccccccc', 'ddddddddddddddddddddd']]
>>> pprint.pprint(stuff, depth=3)
['aaaaaaaaaa',
 ('spam', ('eggs', (...))),
 ['aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa', 'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'],
 ['cccccccccccccccccccc', 'ddddddddddddddddddddd']]
>>> pprint.pprint(stuff, width=60)
['aaaaaaaaaa',
 ('spam',
  ('eggs',
   ('lumberjack',
    ('knights',
     ('ni', ('dead', ('parrot', ('fresh fruit',)))))))),
 ['aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa',
  'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'],
 ['cccccccccccccccccccc', 'ddddddddddddddddddddd']]
```

8.19 repr — Alternate repr() implementation

Note: The repr module has been renamed to reprlib in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/repr.py

The repr module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function:

class repr.Repr
: Class which provides formatting services useful in implementing functions similar to the built-in repr(); size limits for different object types are added to avoid the generation of representations which are excessively long.

repr.aRepr
: This is an instance of Repr which is used to provide the repr() function described below. Changing the attributes of this object will affect the size limits used by repr() and the Python debugger.

repr.repr(obj)
: This is the repr() method of aRepr. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.

8.19.1 Repr Objects

Repr instances provide several attributes which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

Repr.maxlevel
: Depth limit on the creation of recursive representations. The default is 6.

Repr.maxdict
Repr.maxlist
Repr.maxtuple
Repr.maxset
Repr.maxfrozenset
Repr.maxdeque
Repr.maxarray
: Limits on the number of entries represented for the named object type. The default is 4 for maxdict, 5 for maxarray, and 6 for the others.

  New in version 2.4: maxset, maxfrozenset, and set.

Repr.maxlong
: Maximum number of characters in the representation for a long integer. Digits are dropped from the middle. The default is 40.

Repr.maxstring
: Limit on the number of characters in the representation of the string. Note that the "normal" representation of the string is used as the character source: if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is 30.

Repr.maxother
: This limit is used to control the size of object types for which no specific formatting method is available on the Repr object. It is applied in a similar manner as maxstring. The default is 20.

Repr.repr(obj)
: The equivalent to the built-in repr() that uses the formatting imposed by the instance.

Repr.repr1(obj, level)
: Recursive implementation used by repr(). This uses the type of obj to determine which formatting

method to call, passing it obj and level. The type-specific methods should call repr1() to perform recursive formatting, with level - 1 for the value of level in the recursive call.

Repr.repr_TYPE(obj, level)
: Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, TYPE is replaced by string.join(string.split(type(obj).__name__, '_')). Dispatch to these methods is handled by repr1(). Type-specific methods which need to recursively format a value should call self.repr1(subobj, level - 1).

8.19.2 Subclassing Repr Objects

The use of dynamic dispatching by Repr.repr1() allows subclasses of Repr to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added:

```
import repr as reprlib
import sys

class MyRepr(reprlib.Repr):
    def repr_file(self, obj, level):
        if obj.name in ['<stdin>', '<stdout>', '<stderr>']:
            return obj.name
        else:
            return repr(obj)

aRepr = MyRepr()
print aRepr.repr(sys.stdin)         # prints '<stdin>'
```

CHAPTER

NINE

NUMERIC AND MATHEMATICAL MODULES

The modules described in this chapter provide numeric and math-related functions and data types. The numbers module defines an abstract hierarchy of numeric types. The math and cmath modules contain various mathematical functions for floating-point and complex numbers. For users more interested in decimal accuracy than in speed, the decimal module supports exact representations of decimal numbers.

The following modules are documented in this chapter:

9.1 numbers — Numeric abstract base classes

New in version 2.6.

The numbers module (PEP 3141) defines a hierarchy of numeric abstract base classes which progressively define more operations. None of the types defined in this module can be instantiated.

class numbers.Number
: The root of the numeric hierarchy. If you just want to check if an argument x is a number, without caring what kind, use isinstance(x, Number).

9.1.1 The numeric tower

class numbers.Complex
: Subclasses of this type describe complex numbers and include the operations that work on the built-in complex type. These are: conversions to complex and bool, real, imag, +, -, *, /, abs(), conjugate(), ==, and !=. All except - and != are abstract.

  real
  : Abstract. Retrieves the real component of this number.

  imag
  : Abstract. Retrieves the imaginary component of this number.

  conjugate()
  : Abstract. Returns the complex conjugate. For example, (1+3j).conjugate() == (1-3j).

class numbers.Real
: To Complex, Real adds the operations that work on real numbers.

  In short, those are: a conversion to float, math.trunc(), round(), math.floor(), math.ceil(), divmod(), //, %, <, <=, >, and >=.

  Real also provides defaults for complex(), real, imag, and conjugate().

class numbers.Rational
Subtypes Real and adds numerator and denominator properties, which should be in lowest terms. With these, it provides a default for float().

numerator
Abstract.

denominator
Abstract.

class numbers.Integral
Subtypes Rational and adds a conversion to int. Provides defaults for float(), numerator, and denominator. Adds abstract methods for ** and bit-string operations: <<, >>, &, ^, |, ~.

9.1.2 Notes for type implementors

Implementors should be careful to make equal numbers equal and hash them to the same values. This may be subtle if there are two different extensions of the real numbers. For example, fractions.Fraction implements hash() as follows:

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

Adding More Numeric ABCs

There are, of course, more possible ABCs for numbers, and this would be a poor hierarchy if it precluded the possibility of adding those. You can add MyFoo between Complex and Real with:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

Implementing the arithmetic operations

We want to implement the arithmetic operations so that mixed-mode operations either call an implementation whose author knew about the types of both arguments, or convert both to the nearest built in type and do the operation there. For subtypes of Integral, this means that __add__() and __radd__() should be defined as:

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
```

(continues on next page)

(continued from previous page)

```
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

There are 5 different cases for a mixed-type operation on subclasses of Complex. I'll refer to all of the above code that doesn't refer to MyIntegral and OtherTypeIKnowAbout as "boilerplate". a will be an instance of A, which is a subtype of Complex (a : A <: Complex), and b : B <: Complex. I'll consider a + b:

1. If A defines an __add__() which accepts b, all is well.
2. If A falls back to the boilerplate code, and it were to return a value from __add__(), we'd miss the possibility that B defines a more intelligent __radd__(), so the boilerplate should return NotImplemented from __add__(). (Or A may not implement __add__() at all.)
3. Then B's __radd__() gets a chance. If it accepts a, all is well.
4. If it falls back to the boilerplate, there are no more possible methods to try, so this is where the default implementation should live.
5. If B <: A, Python tries B.__radd__ before A.__add__. This is ok, because it was implemented with knowledge of A, so it can handle those instances before delegating to Complex.

If A <: Complex and B <: Real without sharing any other knowledge, then the appropriate shared operation is the one involving the built in complex, and both __radd__() s land there, so a+b == b+a.

Because most of the operations on any given type will be very similar, it can be useful to define a helper function which generates the forward and reverse instances of any given operator. For example, fractions.Fraction uses:

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, long, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
```

(continues on next page)

(continued from previous page)

```
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 math — Mathematical functions

This module is always available. It provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the cmath module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

9.2.1 Number-theoretic and representation functions

math.ceil(x)
: Return the ceiling of x as a float, the smallest integer value greater than or equal to x.

math.copysign(x, y)
: Return x with the sign of y. On a platform that supports signed zeros, copysign(1.0, -0.0) returns -1.0.

  New in version 2.6.

math.fabs(x)
: Return the absolute value of x.

math.factorial(x)
: Return x factorial. Raises ValueError if x is not integral or is negative.

  New in version 2.6.

math.floor(x)
Return the floor of x as a float, the largest integer value less than or equal to x.

math.fmod(x, y)
Return fmod(x, y), as defined by the platform C library. Note that the Python expression x % y may not return the same result. The intent of the C standard is that fmod(x, y) be exactly (mathematically; to infinite precision) equal to x - n*y for some integer n such that the result has the same sign as x and magnitude less than abs(y). Python's x % y returns a result with the sign of y instead, and may not be exactly computable for float arguments. For example, fmod(-1e-100, 1e100) is -1e-100, but the result of Python's -1e-100 % 1e100 is 1e100-1e-100, which cannot be represented exactly as a float, and rounds to the surprising 1e100. For this reason, function fmod() is generally preferred when working with floats, while Python's x % y is preferred when working with integers.

math.frexp(x)
Return the mantissa and exponent of x as the pair (m, e). m is a float and e is an integer such that x == m * 2**e exactly. If x is zero, returns (0.0, 0), otherwise 0.5 <= abs(m) < 1. This is used to "pick apart" the internal representation of a float in a portable way.

math.fsum(iterable)
Return an accurate floating point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums:

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

The algorithm's accuracy depends on IEEE-754 arithmetic guarantees and the typical case where the rounding mode is half-even. On some non-Windows builds, the underlying C library uses extended precision addition and may occasionally double-round an intermediate sum causing it to be off in its least significant bit.

For further discussion and two alternative approaches, see the ASPN cookbook recipes for accurate floating point summation.

New in version 2.6.

math.isinf(x)
Check if the float x is positive or negative infinity.

New in version 2.6.

math.isnan(x)
Check if the float x is a NaN (not a number). For more information on NaNs, see the IEEE 754 standards.

New in version 2.6.

math.ldexp(x, i)
Return x * (2**i). This is essentially the inverse of function frexp().

math.modf(x)
Return the fractional and integer parts of x. Both results carry the sign of x and are floats.

math.trunc(x)
Return the Real value x truncated to an Integral (usually a long integer). Uses the __trunc__ method.

New in version 2.6.

Note that frexp() and modf() have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an 'output parameter' (there is no such thing in Python).

For the ceil(), floor(), and modf() functions, note that all floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float x with abs(x) >= 2**52 necessarily has no fractional bits.

9.2.2 Power and logarithmic functions

math.exp(x)
Return e**x.

math.expm1(x)
Return e**x - 1. For small floats x, the subtraction in exp(x) - 1 can result in a significant loss of precision; the expm1() function provides a way to compute this quantity to full precision:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

New in version 2.7.

math.log(x[, base])
With one argument, return the natural logarithm of x (to base e).

With two arguments, return the logarithm of x to the given base, calculated as log(x)/log(base).

Changed in version 2.3: base argument added.

math.log1p(x)
Return the natural logarithm of 1+x (base e). The result is calculated in a way which is accurate for x near zero.

New in version 2.6.

math.log10(x)
Return the base-10 logarithm of x. This is usually more accurate than log(x, 10).

math.pow(x, y)
Return x raised to the power y. Exceptional cases follow Annex 'F' of the C99 standard as far as possible. In particular, pow(1.0, x) and pow(x, 0.0) always return 1.0, even when x is a zero or a NaN. If both x and y are finite, x is negative, and y is not an integer then pow(x, y) is undefined, and raises ValueError.

Unlike the built-in ** operator, math.pow() converts both its arguments to type float. Use ** or the built-in pow() function for computing exact integer powers.

Changed in version 2.6: The outcome of 1**nan and nan**0 was undefined.

math.sqrt(x)
Return the square root of x.

9.2.3 Trigonometric functions

math.acos(x)
Return the arc cosine of x, in radians.

math.asin(x)
Return the arc sine of x, in radians.

math.atan(x)
Return the arc tangent of x, in radians.

math.atan2(y, x)
Return atan(y / x), in radians. The result is between -pi and pi. The vector in the plane from the origin to point (x, y) makes this angle with the positive X axis. The point of atan2() is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, atan(1) and atan2(1, 1) are both pi/4, but atan2(-1, -1) is -3*pi/4.

math.cos(x)
Return the cosine of x radians.

math.hypot(x, y)
Return the Euclidean norm, sqrt(x*x + y*y). This is the length of the vector from the origin to point (x, y).

math.sin(x)
Return the sine of x radians.

math.tan(x)
Return the tangent of x radians.

9.2.4 Angular conversion

math.degrees(x)
Convert angle x from radians to degrees.

math.radians(x)
Convert angle x from degrees to radians.

9.2.5 Hyperbolic functions

math.acosh(x)
Return the inverse hyperbolic cosine of x.

New in version 2.6.

math.asinh(x)
Return the inverse hyperbolic sine of x.

New in version 2.6.

math.atanh(x)
Return the inverse hyperbolic tangent of x.

New in version 2.6.

math.cosh(x)
Return the hyperbolic cosine of x.

math.sinh(x)
Return the hyperbolic sine of x.

math.tanh(x)
Return the hyperbolic tangent of x.

9.2.6 Special functions

math.erf(x)
Return the error function at x.

New in version 2.7.

math.erfc(x)
Return the complementary error function at x.

New in version 2.7.

math.gamma(x)
Return the Gamma function at x.

New in version 2.7.

math.lgamma(x)
Return the natural logarithm of the absolute value of the Gamma function at x.

New in version 2.7.

9.2.7 Constants

math.pi
The mathematical constant $\pi = 3.141592\ldots$, to available precision.

math.e
The mathematical constant $e = 2.718281\ldots$, to available precision.

CPython implementation detail: The math module consists mostly of thin wrappers around the platform C math library functions. Behavior in exceptional cases follows Annex F of the C99 standard where appropriate. The current implementation will raise ValueError for invalid operations like sqrt(-1.0) or log(0.0) (where C99 Annex F recommends signaling invalid operation or divide-by-zero), and OverflowError for results that overflow (for example, exp(1000.0)). A NaN will not be returned from any of the functions above unless one or more of the input arguments was a NaN; in that case, most functions will return a NaN, but (again following C99 Annex F) there are some exceptions to this rule, for example pow(float('nan'), 0.0) or hypot(float('nan'), float('inf')).

Note that Python makes no effort to distinguish signaling NaNs from quiet NaNs, and behavior for signaling NaNs remains unspecified. Typical behavior is to treat all NaNs as though they were quiet.

Changed in version 2.6: Behavior in special cases now aims to follow C99 Annex F. In earlier versions of Python the behavior in special cases was loosely specified.

See also:

Module cmath Complex number versions of many of these functions.

9.3 cmath — Mathematical functions for complex numbers

This module is always available. It provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. They will also accept any Python object that has either a __complex__() or a __float__() method: these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

Note: On platforms with hardware and system-level support for signed zeros, functions involving branch cuts are continuous on both sides of the branch cut: the sign of the zero distinguishes one side of the branch cut from the other. On platforms that do not support signed zeros the continuity is as specified below.

9.3.1 Conversions to and from polar coordinates

A Python complex number z is stored internally using rectangular or Cartesian coordinates. It is completely determined by its real part z.real and its imaginary part z.imag. In other words:

```
z == z.real + z.imag*1j
```

Polar coordinates give an alternative way to represent a complex number. In polar coordinates, a complex number z is defined by the modulus r and the phase angle phi. The modulus r is the distance from z to the origin, while the phase phi is the counterclockwise angle, measured in radians, from the positive x-axis to the line segment that joins the origin to z.

The following functions can be used to convert from the native rectangular coordinates to polar coordinates and back.

cmath.phase(x)
: Return the phase of x (also known as the argument of x), as a float. phase(x) is equivalent to math.atan2(x.imag, x.real). The result lies in the range [-π, π], and the branch cut for this operation lies along the negative real axis, continuous from above. On systems with support for signed zeros (which includes most systems in current use), this means that the sign of the result is the same as the sign of x.imag, even when x.imag is zero:

```
>>> phase(complex(-1.0, 0.0))
3.1415926535897931
>>> phase(complex(-1.0, -0.0))
-3.1415926535897931
```

: New in version 2.6.

Note: The modulus (absolute value) of a complex number x can be computed using the built-in abs() function. There is no separate cmath module function for this operation.

cmath.polar(x)
: Return the representation of x in polar coordinates. Returns a pair (r, phi) where r is the modulus of x and phi is the phase of x. polar(x) is equivalent to (abs(x), phase(x)).

: New in version 2.6.

cmath.rect(r, phi)
: Return the complex number x with polar coordinates r and phi. Equivalent to r * (math.cos(phi) + math.sin(phi)*1j).

: New in version 2.6.

9.3.2 Power and logarithmic functions

cmath.exp(x)
: Return the exponential value e**x.

cmath.log(x[, base])
Returns the logarithm of x to the given base. If the base is not specified, returns the natural logarithm of x. There is one branch cut, from 0 along the negative real axis to -∞, continuous from above.

Changed in version 2.4: base argument added.

cmath.log10(x)
Return the base-10 logarithm of x. This has the same branch cut as log().

cmath.sqrt(x)
Return the square root of x. This has the same branch cut as log().

9.3.3 Trigonometric functions

cmath.acos(x)
Return the arc cosine of x. There are two branch cuts: One extends right from 1 along the real axis to ∞, continuous from below. The other extends left from -1 along the real axis to -∞, continuous from above.

cmath.asin(x)
Return the arc sine of x. This has the same branch cuts as acos().

cmath.atan(x)
Return the arc tangent of x. There are two branch cuts: One extends from 1j along the imaginary axis to ∞j, continuous from the right. The other extends from -1j along the imaginary axis to -∞j, continuous from the left.

Changed in version 2.6: direction of continuity of upper cut reversed

cmath.cos(x)
Return the cosine of x.

cmath.sin(x)
Return the sine of x.

cmath.tan(x)
Return the tangent of x.

9.3.4 Hyperbolic functions

cmath.acosh(x)
Return the inverse hyperbolic cosine of x. There is one branch cut, extending left from 1 along the real axis to -∞, continuous from above.

cmath.asinh(x)
Return the inverse hyperbolic sine of x. There are two branch cuts: One extends from 1j along the imaginary axis to ∞j, continuous from the right. The other extends from -1j along the imaginary axis to -∞j, continuous from the left.

Changed in version 2.6: branch cuts moved to match those recommended by the C99 standard

cmath.atanh(x)
Return the inverse hyperbolic tangent of x. There are two branch cuts: One extends from 1 along the real axis to ∞, continuous from below. The other extends from -1 along the real axis to -∞, continuous from above.

Changed in version 2.6: direction of continuity of right cut reversed

cmath.cosh(x)
Return the hyperbolic cosine of x.

cmath.sinh(x)
Return the hyperbolic sine of x.

cmath.tanh(x)
Return the hyperbolic tangent of x.

9.3.5 Classification functions

cmath.isinf(x)
Return True if the real or the imaginary part of x is positive or negative infinity.

New in version 2.6.

cmath.isnan(x)
Return True if the real or imaginary part of x is not a number (NaN).

New in version 2.6.

9.3.6 Constants

cmath.pi
The mathematical constant π, as a float.

cmath.e
The mathematical constant e, as a float.

Note that the selection of functions is similar, but not identical, to that in module math. The reason for having two modules is that some users aren't interested in complex numbers, and perhaps don't even know what they are. They would rather have math.sqrt(-1) raise an exception than return a complex number. Also note that the functions defined in cmath always return a complex number, even if the answer can be expressed as a real number (in which case the complex number has an imaginary part of zero).

A note on branch cuts: They are curves along which the given function fails to be continuous. They are a necessary feature of many complex functions. It is assumed that if you need to compute with complex functions, you will understand about branch cuts. Consult almost any (not too elementary) book on complex variables for enlightenment. For information of the proper choice of branch cuts for numerical purposes, a good reference should be the following:

See also:

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothing's sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165–211.

9.4 decimal — Decimal fixed point and floating point arithmetic

New in version 2.4.

The decimal module provides support for decimal floating point arithmetic. It offers several advantages over the float datatype:

- Decimal "is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school." – excerpt from the decimal arithmetic specification.

- Decimal numbers can be represented exactly. In contrast, numbers like 1.1 and 2.2 do not have exact representations in binary floating point. End users typically would not expect 1.1 + 2.2 to display as 3.3000000000000003 as it does with binary floating point.
- The exactness carries over into arithmetic. In decimal floating point, 0.1 + 0.1 + 0.1 - 0.3 is exactly equal to zero. In binary floating point, the result is 5.5511151231257827e-017. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.
- The decimal module incorporates a notion of significant places so that 1.30 + 1.20 is 2.50. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the "schoolbook" approach uses all the figures in the multiplicands. For instance, 1.3 * 1.2 gives 1.56 while 1.30 * 1.20 gives 1.5600.
- Unlike hardware based binary floating point, the decimal module has a user alterable precision (defaulting to 28 places) which can be as large as needed for a given problem:

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling. This includes an option to enforce exact arithmetic by using exceptions to block any inexact operations.
- The decimal module was designed to support "without prejudice, both exact unrounded decimal arithmetic (sometimes called fixed-point arithmetic) and rounded floating-point arithmetic." – excerpt from the decimal arithmetic specification.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as Infinity, -Infinity, and NaN. The standard also differentiates -0 from +0.

The context for arithmetic is an environment specifying precision, rounding rules, limits on exponents, flags indicating the results of operations, and trap enablers which determine whether signals are treated as exceptions. Rounding options include ROUND_CEILING, ROUND_DOWN, ROUND_FLOOR, ROUND_HALF_DOWN, ROUND_HALF_EVEN, ROUND_HALF_UP, ROUND_UP, and ROUND_05UP.

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: Clamped, InvalidOperation, DivisionByZero, Inexact, Rounded, Subnormal, Overflow, and Underflow.

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is set to one, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

See also:

- IBM's General Decimal Arithmetic Specification, The General Decimal Arithmetic Specification.

9.4.1 Quick-start Tutorial

The usual start to using decimals is importing the module, viewing the current context with getcontext() and, if necessary, setting new values for precision, rounding, or enabled traps:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
        capitals=1, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

Decimal instances can be constructed from integers, strings, floats, or tuples. Construction from an integer or a float performs an exact conversion of the value of that integer or float. Decimal numbers include special values such as NaN which stands for "Not a number", positive and negative Infinity, and -0.

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.41421356237')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.4142135623730950488016887242')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

The significance of a new Decimal is determined solely by the number of digits input. Context precision and rounding only come into play during arithmetic operations.

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

Decimals interact well with much of the rest of Python. Here is a small decimal floating point flying circus:

```
>>> data = map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split())
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
```

(continues on next page)

(continued from previous page)

```
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)     # round() first converts to binary floating point
1.3
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

And some mathematical functions are also available to Decimal:

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

The quantize() method rounds a number to a fixed exponent. This method is useful for monetary applications that often round results to a fixed number of places:

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

As shown above, the getcontext() function accesses the current context and allows the settings to be changed. This approach meets the needs of most applications.

For more advanced work, it may be useful to create alternate contexts using the Context() constructor. To make an alternate active, use the setcontext() function.

In accordance with the standard, the decimal module provides two ready to use standard contexts, BasicContext and ExtendedContext. The former is especially useful for debugging because many of the traps are enabled:

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
```

(continues on next page)

(continued from previous page)

```
        capitals=1, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

Contexts also have signal flags for monitoring exceptional conditions encountered during computations. The flags remain set until explicitly cleared, so it is best to clear the flags before each set of monitored computations by using the clear_flags() method.

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
        capitals=1, flags=[Rounded, Inexact], traps=[])
```

The flags entry shows that the rational approximation to Pi was rounded (digits beyond the context precision were thrown away) and that the result is inexact (some of the discarded digits were non-zero).

Individual traps are set using the dictionary in the traps field of a context:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

Most programs adjust the current context only once, at the beginning of the program. And, in many applications, data is converted to Decimal with a single cast inside a loop. With context set and decimals created, the bulk of the program manipulates the data no differently than with other Python numeric types.

9.4.2 Decimal objects

class decimal.Decimal([value[, context]])

Construct a new Decimal object based from value.

value can be an integer, string, tuple, float, or another Decimal object. If no value is given, returns Decimal('0'). If value is a string, it should conform to the decimal numeric string syntax after leading and trailing whitespace characters are removed:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

If value is a unicode string then other Unicode decimal digits are also permitted where digit appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanāgarī digits) along with the fullwidth digits u'\uff10' through u'\uff19'.

If value is a tuple, it should have three components, a sign (0 for positive or 1 for negative), a tuple of digits, and an integer exponent. For example, Decimal((0, (1, 4, 1, 4), -3)) returns Decimal('1.414').

If value is a float, the binary floating point value is losslessly converted to its exact decimal equivalent. This conversion can often require 53 or more digits of precision. For example, Decimal(float('1.1')) converts to Decimal('1.100000000000000088817841970012523233890533447265625').

The context precision does not affect how many digits are stored. That is determined exclusively by the number of digits in value. For example, Decimal('3.00000') records all five zeros even if the context precision is only three.

The purpose of the context argument is determining what to do if value is a malformed string. If the context traps InvalidOperation, an exception is raised; otherwise, the constructor returns a new Decimal with the value of NaN.

Once constructed, Decimal objects are immutable.

Changed in version 2.6: leading and trailing whitespace characters are permitted when creating a Decimal instance from a string.

Changed in version 2.7: The argument to the constructor is now permitted to be a float instance.

Decimal floating point objects share many properties with the other built-in numeric types such as float and int. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as float or long).

There are some small differences between arithmetic on Decimal objects and arithmetic on integers and floats. When the remainder operator % is applied to Decimal objects, the sign of the result is the sign of the dividend rather than the sign of the divisor:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

The integer division operator // behaves analogously, returning the integer part of the true quotient (truncating towards zero) rather than its floor, so as to preserve the usual identity x == (x // y) * y + x % y:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

The % and // operators implement the remainder and divide-integer operations (respectively) as described in the specification.

Decimal objects cannot generally be combined with floats in arithmetic operations: an attempt to add a Decimal to a float, for example, will raise a TypeError. There's one exception to this rule: it's possible to use Python's comparison operators to compare a float instance x with a Decimal instance y. Without this exception, comparisons between Decimal and float instances would follow the general rules for comparing objects of different types described in the expressions section of the reference manual, leading to confusing results.

Changed in version 2.7: A comparison between a float instance x and a Decimal instance y now returns a result based on the values of x and y. In earlier versions x < y returned the same (arbitrary) result for any Decimal instance x and any float instance y.

In addition to the standard numeric properties, decimal floating point objects also have a number of specialized methods:

adjusted()
: Return the adjusted exponent after shifting out the coefficient's rightmost digits until only the lead digit remains: Decimal('321e+5').adjusted() returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

as_tuple()
: Return a named tuple representation of the number: DecimalTuple(sign, digits, exponent).

  Changed in version 2.6: Use a named tuple.

canonical()
: Return the canonical encoding of the argument. Currently, the encoding of a Decimal instance is always canonical, so this operation returns its argument unchanged.

  New in version 2.6.

compare(other[, context])
: Compare the values of two Decimal instances. This operation behaves in the same way as the usual comparison method __cmp__(), except that compare() returns a Decimal instance rather than an integer, and if either operand is a NaN then the result is a NaN:

```
a or b is a NaN ==> Decimal('NaN')
a < b           ==> Decimal('-1')
a == b          ==> Decimal('0')
a > b           ==> Decimal('1')
```

compare_signal(other[, context])
: This operation is identical to the compare() method, except that all NaNs signal. That is, if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN.

  New in version 2.6.

compare_total(other)
: Compare two operands using their abstract representation rather than their numerical value. Similar to the compare() method, but the result gives a total ordering on Decimal instances. Two Decimal instances with the same numeric value but different representations compare unequal in this ordering:

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Quiet and signaling NaNs are also included in the total ordering. The result of this function is Decimal('0') if both operands have the same representation, Decimal('-1') if the first operand is lower in the total order than the second, and Decimal('1') if the first operand is higher in the total order than the second operand. See the specification for details of the total order.

New in version 2.6.

compare_total_mag(other)
Compare two operands using their abstract representation rather than their value as in compare_total(), but ignoring the sign of each operand. x.compare_total_mag(y) is equivalent to x.copy_abs().compare_total(y.copy_abs()).

New in version 2.6.

conjugate()
Just returns self, this method is only to comply with the Decimal Specification.

New in version 2.6.

copy_abs()
Return the absolute value of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

New in version 2.6.

copy_negate()
Return the negation of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

New in version 2.6.

copy_sign(other)
Return a copy of the first operand with the sign set to be the same as the sign of the second operand. For example:

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

New in version 2.6.

exp([context])
Return the value of the (natural) exponential function e**x at the given number. The result is correctly rounded using the ROUND_HALF_EVEN rounding mode.

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

New in version 2.6.

from_float(f)
Classmethod that converts a float to a decimal number, exactly.

Note Decimal.from_float(0.1) is not the same as Decimal('0.1'). Since 0.1 is not exactly representable in binary floating point, the value is stored as the nearest representable value which is 0x1.999999999999ap-4. That equivalent value in decimal is 0.1000000000000000055511151231257827021181583404541015625.

Note: From Python 2.7 onwards, a Decimal instance can also be constructed directly from a float.

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

New in version 2.7.

fma(other, third[, context])
Fused multiply-add. Return self*other+third with no rounding of the intermediate product self*other.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

New in version 2.6.

is_canonical()
Return True if the argument is canonical and False otherwise. Currently, a Decimal instance is always canonical, so this operation always returns True.

New in version 2.6.

is_finite()
Return True if the argument is a finite number, and False if the argument is an infinity or a NaN.

New in version 2.6.

is_infinite()
Return True if the argument is either positive or negative infinity and False otherwise.

New in version 2.6.

is_nan()
Return True if the argument is a (quiet or signaling) NaN and False otherwise.

New in version 2.6.

is_normal()
Return True if the argument is a normal finite non-zero number with an adjusted exponent greater than or equal to Emin. Return False if the argument is zero, subnormal, infinite or a NaN. Note, the term normal is used here in a different sense with the normalize() method which is used to create canonical values.

New in version 2.6.

is_qnan()
Return True if the argument is a quiet NaN, and False otherwise.

New in version 2.6.

is_signed()
Return True if the argument has a negative sign and False otherwise. Note that zeros and NaNs can both carry signs.

New in version 2.6.

is_snan()
Return True if the argument is a signaling NaN and False otherwise.

New in version 2.6.

is_subnormal()
Return True if the argument is subnormal, and False otherwise. A number is subnormal is if it is nonzero, finite, and has an adjusted exponent less than Emin.

New in version 2.6.

is_zero()
Return True if the argument is a (positive or negative) zero and False otherwise.

New in version 2.6.

ln([context])
Return the natural (base e) logarithm of the operand. The result is correctly rounded using the ROUND_HALF_EVEN rounding mode.

New in version 2.6.

log10([context])
Return the base ten logarithm of the operand. The result is correctly rounded using the ROUND_HALF_EVEN rounding mode.

New in version 2.6.

logb([context])
For a nonzero number, return the adjusted exponent of its operand as a Decimal instance. If the operand is a zero then Decimal('-Infinity') is returned and the DivisionByZero flag is raised. If the operand is an infinity then Decimal('Infinity') is returned.

New in version 2.6.

logical_and(other[, context])
logical_and() is a logical operation which takes two logical operands (see Logical operands). The result is the digit-wise and of the two operands.

New in version 2.6.

logical_invert([context])
logical_invert() is a logical operation. The result is the digit-wise inversion of the operand.

New in version 2.6.

logical_or(other[, context])
logical_or() is a logical operation which takes two logical operands (see Logical operands). The result is the digit-wise or of the two operands.

New in version 2.6.

logical_xor(other[, context])
logical_xor() is a logical operation which takes two logical operands (see Logical operands). The result is the digit-wise exclusive or of the two operands.

New in version 2.6.

max(other[, context])
Like max(self, other) except that the context rounding rule is applied before returning and that NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

max_mag(other[, context])
Similar to the max() method, but the comparison is done using the absolute values of the operands.

New in version 2.6.

min(other[, context])
Like min(self, other) except that the context rounding rule is applied before returning and that NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

min_mag(other[, context])
Similar to the min() method, but the comparison is done using the absolute values of the operands.

New in version 2.6.

next_minus([context])
Return the largest number representable in the given context (or in the current thread's context if no context is given) that is smaller than the given operand.

New in version 2.6.

next_plus([context])
Return the smallest number representable in the given context (or in the current thread's context if no context is given) that is larger than the given operand.

New in version 2.6.

next_toward(other[, context])
If the two operands are unequal, return the number closest to the first operand in the direction of the second operand. If both operands are numerically equal, return a copy of the first operand with the sign set to be the same as the sign of the second operand.

New in version 2.6.

normalize([context])
Normalize the number by stripping the rightmost trailing zeros and converting any result equal to Decimal('0') to Decimal('0e0'). Used for producing canonical values for attributes of an equivalence class. For example, Decimal('32.100') and Decimal('0.321000e+2') both normalize to the equivalent value Decimal('32.1').

number_class([context])
Return a string describing the class of the operand. The returned value is one of the following ten strings.

- "-Infinity", indicating that the operand is negative infinity.
- "-Normal", indicating that the operand is a negative normal number.
- "-Subnormal", indicating that the operand is negative and subnormal.
- "-Zero", indicating that the operand is a negative zero.
- "+Zero", indicating that the operand is a positive zero.
- "+Subnormal", indicating that the operand is positive and subnormal.
- "+Normal", indicating that the operand is a positive normal number.
- "+Infinity", indicating that the operand is positive infinity.
- "NaN", indicating that the operand is a quiet NaN (Not a Number).
- "sNaN", indicating that the operand is a signaling NaN.

New in version 2.6.

quantize(exp[, rounding[, context[, watchexp]]])
Return a value equal to the first operand after rounding and having the exponent of the second operand.

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an InvalidOperation is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand.

Also unlike other operations, quantize never signals Underflow, even if the result is subnormal and inexact.

If the exponent of the second operand is larger than that of the first then rounding may be necessary. In this case, the rounding mode is determined by the rounding argument if given, else by the given context argument; if neither argument is given the rounding mode of the current thread's context is used.

If watchexp is set (default), then an error is returned whenever the resulting exponent is greater than Emax or less than Etiny.

radix()
Return Decimal(10), the radix (base) in which the Decimal class does all its arithmetic. Included for compatibility with the specification.

New in version 2.6.

remainder_near(other[, context])
Return the remainder from dividing self by other. This differs from self % other in that the sign of the remainder is chosen so as to minimize its absolute value. More precisely, the return value is self - n * other where n is the integer nearest to the exact value of self / other, and if two integers are equally near then the even one is chosen.

If the result is zero then its sign will be the sign of self.

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(other[, context])
Return the result of rotating the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to rotate. If the second operand is positive then rotation is to the left; otherwise rotation is to the right. The coefficient of the first operand is padded on the left with zeros to length precision if necessary. The sign and exponent of the first operand are unchanged.

New in version 2.6.

same_quantum(other[, context])
Test whether self and other have the same exponent or whether both are NaN.

scaleb(other[, context])
Return the first operand with exponent adjusted by the second. Equivalently, return the first operand multiplied by 10**other. The second operand must be an integer.

New in version 2.6.

shift(other[, context])
: Return the result of shifting the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to shift. If the second operand is positive then the shift is to the left; otherwise the shift is to the right. Digits shifted into the coefficient are zeros. The sign and exponent of the first operand are unchanged.

  New in version 2.6.

sqrt([context])
: Return the square root of the argument to full precision.

to_eng_string([context])
: Convert to a string, using engineering notation if an exponent is needed.

  Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

  For example, this converts Decimal('123E+1') to Decimal('1.23E+3').

to_integral([rounding[, context]])
: Identical to the to_integral_value() method. The to_integral name has been kept for compatibility with older versions.

to_integral_exact([rounding[, context]])
: Round to the nearest integer, signaling Inexact or Rounded as appropriate if rounding occurs. The rounding mode is determined by the rounding parameter if given, else by the given context. If neither parameter is given then the rounding mode of the current context is used.

  New in version 2.6.

to_integral_value([rounding[, context]])
: Round to the nearest integer without signaling Inexact or Rounded. If given, applies rounding; otherwise, uses the rounding method in either the supplied context or the current context.

  Changed in version 2.6: renamed from to_integral to to_integral_value. The old name remains valid for compatibility.

Logical operands

The logical_and(), logical_invert(), logical_or(), and logical_xor() methods expect their arguments to be logical operands. A logical operand is a Decimal instance whose exponent and sign are both zero, and whose digits are all either 0 or 1.

9.4.3 Context objects

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the getcontext() and setcontext() functions:

decimal.getcontext()
: Return the current context for the active thread.

decimal.setcontext(c)
: Set the current context for the active thread to c.

Beginning with Python 2.5, you can also use the with statement and the localcontext() function to temporarily change the active context.

decimal.localcontext([c])
Return a context manager that will set the current context for the active thread to a copy of c on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used.

New in version 2.5.

For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context:

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
    s = calculate_something()
s = +s  # Round the final result back to the default precision

with localcontext(BasicContext):      # temporarily use the BasicContext
    print Decimal(1) / Decimal(7)
    print Decimal(355) / Decimal(113)
```

New contexts can also be created using the Context constructor described below. In addition, the module provides three pre-made contexts:

class decimal.BasicContext
This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to ROUND_HALF_UP. All flags are cleared. All traps are enabled (treated as exceptions) except Inexact, Rounded, and Subnormal.

Because many of the traps are enabled, this context is useful for debugging.

class decimal.ExtendedContext
This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to ROUND_HALF_EVEN. All flags are cleared. No traps are enabled (so that exceptions are not raised during computations).

Because the traps are disabled, this context is useful for applications that prefer to have result value of NaN or Infinity instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

class decimal.DefaultContext
This context is used by the Context constructor as a prototype for new contexts. Changing a field (such a precision) has the effect of changing the default for new contexts created by the Context constructor.

This context is most useful in multi-threaded environments. Changing one of the fields before threads are started has the effect of setting system-wide defaults. Changing the fields after threads have started is not recommended as it would require thread synchronization to prevent race conditions.

In single threaded environments, it is preferable to not use this context at all. Instead, simply create contexts explicitly as described below.

The default values are precision=28, rounding=ROUND_HALF_EVEN, and enabled traps for Overflow, InvalidOperation, and DivisionByZero.

In addition to the three supplied contexts, new contexts can be created with the Context constructor.

class decimal.Context(prec=None, rounding=None, traps=None, flags=None, Emin=None, Emax=None, capitals=1)

Creates a new context. If a field is not specified or is None, the default values are copied from the DefaultContext. If the flags field is not specified or is None, all flags are cleared.

The prec field is a positive integer that sets the precision for arithmetic operations in the context.

The rounding option is one of:

- ROUND_CEILING (towards Infinity),
- ROUND_DOWN (towards zero),
- ROUND_FLOOR (towards -Infinity),
- ROUND_HALF_DOWN (to nearest with ties going towards zero),
- ROUND_HALF_EVEN (to nearest with ties going to nearest even integer),
- ROUND_HALF_UP (to nearest with ties going away from zero), or
- ROUND_UP (away from zero).
- ROUND_05UP (away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise towards zero)

The traps and flags fields list any signals to be set. Generally, new contexts should only set traps and leave the flags clear.

The Emin and Emax fields are integers specifying the outer limits allowable for exponents.

The capitals field is either 0 or 1 (the default). If set to 1, exponents are printed with a capital E; otherwise, a lowercase e is used: Decimal('6.02e+23').

Changed in version 2.6: The ROUND_05UP rounding mode was added.

The Context class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the Decimal methods described above (with the exception of the adjusted() and as_tuple() methods) there is a corresponding Context method. For example, for a Context instance C and Decimal instance x, C.exp(x) is equivalent to x.exp(context=C). Each Context method accepts a Python integer (an instance of int or long) anywhere that a Decimal instance is accepted.

clear_flags()
: Resets all of the flags to 0.

copy()
: Return a duplicate of the context.

copy_decimal(num)
: Return a copy of the Decimal instance num.

create_decimal(num)
: Creates a new Decimal instance from num but using self as context. Unlike the Decimal constructor, the context precision, rounding method, flags, and traps are applied to the conversion.

  This is useful because constants are often given to a greater precision than is needed by the application. Another benefit is that rounding immediately eliminates unintended effects from digits beyond the current precision. In the following example, using unrounded inputs means that adding zero to a sum can change the result:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

This method implements the to-number operation of the IBM specification. If the argument is a string, no leading or trailing whitespace is permitted.

create_decimal_from_float(f)
Creates a new Decimal instance from a float f but rounding using self as the context. Unlike the Decimal.from_float() class method, the context precision, rounding method, flags, and traps are applied to the conversion.

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
Inexact: None
```

New in version 2.7.

Etiny()
Returns a value equal to Emin - prec + 1 which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to Etiny.

Etop()
Returns a value equal to Emax - prec + 1.

The usual approach to working with decimals is to create Decimal instances and then apply arithmetic operations which take place within the current context for the active thread. An alternative approach is to use context methods for calculating within a specific context. The methods are similar to those for the Decimal class and are only briefly recounted here.

abs(x)
Returns the absolute value of x.

add(x, y)
Return the sum of x and y.

canonical(x)
Returns the same Decimal object x.

compare(x, y)
Compares x and y numerically.

compare_signal(x, y)
Compares the values of the two operands numerically.

compare_total(x, y)
Compares two operands using their abstract representation.

compare_total_mag(x, y)
Compares two operands using their abstract representation, ignoring sign.

copy_abs(x)
Returns a copy of x with the sign set to 0.

copy_negate(x)
Returns a copy of x with the sign inverted.

copy_sign(x, y)
Copies the sign from y to x.

divide(x, y)
: Return x divided by y.

divide_int(x, y)
: Return x divided by y, truncated to an integer.

divmod(x, y)
: Divides two numbers and returns the integer part of the result.

exp(x)
: Returns e ** x.

fma(x, y, z)
: Returns x multiplied by y, plus z.

is_canonical(x)
: Returns True if x is canonical; otherwise returns False.

is_finite(x)
: Returns True if x is finite; otherwise returns False.

is_infinite(x)
: Returns True if x is infinite; otherwise returns False.

is_nan(x)
: Returns True if x is a qNaN or sNaN; otherwise returns False.

is_normal(x)
: Returns True if x is a normal number; otherwise returns False.

is_qnan(x)
: Returns True if x is a quiet NaN; otherwise returns False.

is_signed(x)
: Returns True if x is negative; otherwise returns False.

is_snan(x)
: Returns True if x is a signaling NaN; otherwise returns False.

is_subnormal(x)
: Returns True if x is subnormal; otherwise returns False.

is_zero(x)
: Returns True if x is a zero; otherwise returns False.

ln(x)
: Returns the natural (base e) logarithm of x.

log10(x)
: Returns the base 10 logarithm of x.

logb(x)
: Returns the exponent of the magnitude of the operand's MSD.

logical_and(x, y)
: Applies the logical operation and between each operand's digits.

logical_invert(x)
: Invert all the digits in x.

logical_or(x, y)
: Applies the logical operation or between each operand's digits.

logical_xor(x, y)
: Applies the logical operation xor between each operand's digits.

max(x, y)
: Compares two values numerically and returns the maximum.

max_mag(x, y)
: Compares the values numerically with their sign ignored.

min(x, y)
: Compares two values numerically and returns the minimum.

min_mag(x, y)
: Compares the values numerically with their sign ignored.

minus(x)
: Minus corresponds to the unary prefix minus operator in Python.

multiply(x, y)
: Return the product of x and y.

next_minus(x)
: Returns the largest representable number smaller than x.

next_plus(x)
: Returns the smallest representable number larger than x.

next_toward(x, y)
: Returns the number closest to x, in direction towards y.

normalize(x)
: Reduces x to its simplest form.

number_class(x)
: Returns an indication of the class of x.

plus(x)
: Plus corresponds to the unary prefix plus operator in Python. This operation applies the context precision and rounding, so it is not an identity operation.

power(x, y[, modulo])
: Return x to the power of y, reduced modulo modulo if given.

  With two arguments, compute x**y. If x is negative then y must be integral. The result will be inexact unless y is integral and the result is finite and can be expressed exactly in 'precision' digits. The result should always be correctly rounded, using the rounding mode of the current thread's context.

  With three arguments, compute (x**y) % modulo. For the three argument form, the following restrictions on the arguments hold:

  - all three arguments must be integral
  - y must be nonnegative
  - at least one of x or y must be nonzero
  - modulo must be nonzero and have at most 'precision' digits

  The value resulting from Context.power(x, y, modulo) is equal to the value that would be obtained by computing (x**y) % modulo with unbounded precision, but is computed more efficiently. The exponent of the result is zero, regardless of the exponents of x, y and modulo. The result is always exact.

Changed in version 2.6: y may now be nonintegral in x**y. Stricter requirements for the three-argument version.

quantize(x, y)
Returns a value equal to x (rounded), having the exponent of y.

radix()
Just returns 10, as this is Decimal, :)

remainder(x, y)
Returns the remainder from integer division.

The sign of the result, if non-zero, is the same as that of the original dividend.

remainder_near(x, y)
Returns x - y * n, where n is the integer nearest the exact value of x / y (if the result is 0 then its sign will be the sign of x).

rotate(x, y)
Returns a rotated copy of x, y times.

same_quantum(x, y)
Returns True if the two operands have the same exponent.

scaleb(x, y)
Returns the first operand after adding the second value its exp.

shift(x, y)
Returns a shifted copy of x, y times.

sqrt(x)
Square root of a non-negative number to context precision.

subtract(x, y)
Return the difference between x and y.

to_eng_string(x)
Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

to_integral_exact(x)
Rounds to an integer.

to_sci_string(x)
Converts a number to a string using scientific notation.

9.4.4 Signals

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is set whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context's trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the DivisionByZero trap is set, then a DivisionByZero exception is raised upon encountering the condition.

class decimal.Clamped
Altered an exponent to fit representation constraints.

Typically, clamping occurs when an exponent falls outside the context's Emin and Emax limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.

class decimal.DecimalException
Base class for other signals and a subclass of ArithmeticError.

class decimal.DivisionByZero
Signals the division of a non-infinite number by zero.

Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns Infinity or -Infinity with the sign determined by the inputs to the calculation.

class decimal.Inexact
Indicates that rounding occurred and the result is not exact.

Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.

class decimal.InvalidOperation
An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns NaN. Possible causes include:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
x._rescale( non-integer )
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.Overflow
Numerical overflow.

Indicates the exponent is larger than Emax after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to Infinity. In either case, Inexact and Rounded are also signaled.

class decimal.Rounded
Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding 5.00 to 5.0). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

class decimal.Subnormal
Exponent was lower than Emin prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

class decimal.Underflow
Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. Inexact and Subnormal are also signaled.

The following table summarizes the hierarchy of signals:

```
exceptions.ArithmeticError(exceptions.StandardError)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
```

9.4.5 Floating Point Notes

Mitigating round-off error with increased precision

The use of decimal floating point eliminates decimal representation error (making it possible to represent 0.1 exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

The decimal module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
```

(continues on next page)

(continued from previous page)

```
>>> u * (v+w)
Decimal('0.0060000')
```

Special values

The number system for the decimal module provides special values including NaN, sNaN, -Infinity, Infinity, and two zeros, +0 and -0.

Infinities can be constructed directly with: Decimal('Infinity'). Also, they can arise from dividing by zero when the DivisionByZero signal is not trapped. Likewise, when the Overflow signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return NaN, or if the InvalidOperation signal is trapped, raise an exception. For example, 0/0 returns NaN which means "not a number". This variety of NaN is quiet and, once created, will flow through other computations always resulting in another NaN. This behavior can be useful for a series of computations that occasionally have missing inputs — it allows the calculation to proceed while flagging specific results as invalid.

A variant is sNaN which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python's comparison operators can be a little surprising where a NaN is involved. A test for equality where one of the operands is a quiet or signaling NaN always returns False (even when doing Decimal('NaN')==Decimal('NaN')), while a test for inequality always returns True. An attempt to compare two Decimals using any of the <, <=, > or >= operators will raise the InvalidOperation signal if either operand is a NaN, and return False if this signal is not trapped. Note that the General Decimal Arithmetic specification does not specify the behavior of direct comparisons; these rules for comparisons involving a NaN were taken from the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the compare() and compare-signal() methods instead.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating point representations, it is not immediately obvious that the following calculation returns a value equal to zero:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000000026')
```

9.4.6 Working with threads

The getcontext() function accesses a different Context object for each thread. Having separate thread contexts means that threads may make changes (such as getcontext.prec=10) without interfering with other threads.

Likewise, the setcontext() function automatically assigns its target to the current thread.

If setcontext() has not been called before getcontext(), then getcontext() will automatically create a new context for use in the current thread.

The new context is copied from a prototype context called DefaultContext. To control the defaults so that each thread will use the same values throughout the application, directly modify the DefaultContext object. This should be done before any threads are started so that there won't be a race condition between threads calling getcontext(). For example:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.7 Recipes

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the Decimal class:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or blank
    trailneg:optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = map(str, digits)
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
```

(continues on next page)

(continued from previous page)

```
    for i in range(places):
        build(next() if digits else '0')
    build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print pi()
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input type.

    >>> print exp(Decimal(1))
    2.718281828459045235360287471
    >>> print exp(Decimal(2))
    7.389056098930650227230427461
    >>> print exp(2.0)
    7.38905609893
    >>> print exp(2+0j)
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
```

(continues on next page)

(continued from previous page)

```
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    >>> print cos(Decimal('0.5'))
    0.87758256189037271611628158260
    >>> print cos(0.5)
    0.87758256189
    >>> print cos(0.5+0j)
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    >>> print sin(Decimal('0.5'))
    0.47942553860420300027328793521
    >>> print sin(0.5)
    0.479425538604
    >>> print sin(0.5+0j)
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.8 Decimal FAQ

Q. It is cumbersome to type decimal.Decimal('1234.5'). Is there a way to minimize typing when using the interactive interpreter?

A. Some users abbreviate the constructor to just a single letter:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used?

A. The quantize() method rounds to a fixed number of decimal places. If the Inexact trap is set, it is also useful for validation:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal( '0.01 ')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
  ...
Inexact: None
```

Q. Once I have valid two place inputs, how do I maintain that invariant throughout an application?

A. Some operations like addition, subtraction, and multiplication by an integer will automatically preserve fixed point. Others operations, like division and non-integer multiplication, will change the number of decimal places and need to be followed-up with a quantize() step:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)     # And quantize division
Decimal('0.03')
```

In developing fixed-point applications, it is convenient to define functions to handle the quantize() step:

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                       # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. There are many ways to express the same value. The numbers 200, 200.000, 2E2, and 02E+4 all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value?

A. The normalize() method maps all equivalent values to a single representative:

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing 5.0E+3 as 5000 keeps the value constant but cannot show the original's two-place significance.

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeros, losing significance, but keeping the value unchanged:

```
def remove_exponent(d):
    '''Remove exponent and trailing zeros.

    >>> remove_exponent(Decimal('5E+3'))
    Decimal('5000')

    '''
    return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

Q. Is there a way to convert a regular float to a Decimal?

A. Yes, any binary floating point number can be exactly expressed as a Decimal though an exact conversion may take more precision than intuition would suggest:

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. Within a complex calculation, how can I make sure that I haven't gotten a spurious result because of insufficient precision or rounding anomalies.

A. The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q. I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions?

A. Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that "what you type is what you get". A disadvantage is that the results can look odd if you forget that the inputs haven't been rounded:

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

The solution is either to increase precision or to force rounding of inputs using the unary plus operation:

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

Alternatively, inputs can be rounded upon creation using the Context.create_decimal() method:

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

9.5 fractions — Rational numbers

New in version 2.6.

Source code: Lib/fractions.py

The fractions module provides support for rational number arithmetic.

A Fraction instance can be constructed from a pair of integers, from another rational number, or from a string.

class fractions.Fraction(numerator=0, denominator=1)
class fractions.Fraction(other_fraction)
class fractions.Fraction(float)
class fractions.Fraction(decimal)
class fractions.Fraction(string)

The first version requires that numerator and denominator are instances of numbers.Rational and returns a new Fraction instance with value numerator/denominator. If denominator is 0, it raises a ZeroDivisionError. The second version requires that other_fraction is an instance of numbers.Rational and returns a Fraction instance with the same value. The next two versions accept either a float or a decimal.Decimal instance, and return a Fraction instance with exactly the same value. Note that due to the usual issues with binary floating-point (see tut-fp-issues), the argument to Fraction(1.1) is not exactly equal to 11/10, and so Fraction(1.1) does not return Fraction(11, 10) as one might expect. (But see the documentation for the limit_denominator() method below.) The last version of the constructor expects a string or unicode instance. The usual form for this instance is:

```
[sign] numerator ['/' denominator]
```

where the optional sign may be either '+' or '-' and numerator and denominator (if present) are strings of decimal digits. In addition, any string that represents a finite value and is accepted by the float constructor is also accepted by the Fraction constructor. In either form the input string may also have leading and/or trailing whitespace. Here are some examples:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
```

(continues on next page)

(continued from previous page)

```
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

The Fraction class inherits from the abstract base class numbers.Rational, and implements all of the methods and operations from that class. Fraction instances are hashable, and should be treated as immutable. In addition, Fraction has the following methods:

Changed in version 2.7: The Fraction constructor now accepts float and decimal.Decimal instances.

from_float(flt)
: This class method constructs a Fraction representing the exact value of flt, which must be a float. Beware that Fraction.from_float(0.3) is not the same value as Fraction(3, 10).

  Note: From Python 2.7 onwards, you can also construct a Fraction instance directly from a float.

from_decimal(dec)
: This class method constructs a Fraction representing the exact value of dec, which must be a decimal.Decimal.

  Note: From Python 2.7 onwards, you can also construct a Fraction instance directly from a decimal.Decimal instance.

limit_denominator(max_denominator=1000000)
: Finds and returns the closest Fraction to self that has denominator at most max_denominator. This method is useful for finding rational approximations to a given floating-point number:

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

  or for recovering a rational number that's represented as a float:

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

fractions.gcd(a, b)
: Return the greatest common divisor of the integers a and b. If either a or b is nonzero, then the

absolute value of gcd(a, b) is the largest integer that divides both a and b. gcd(a,b) has the same sign as b if b is nonzero; otherwise it takes the sign of a. gcd(0, 0) returns 0.

See also:

Module numbers The abstract base classes making up the numeric tower.

9.6 random — Generate pseudo-random numbers

Source code: Lib/random.py

This module implements pseudo-random number generators for various distributions.

For integers, uniform selection from a range. For sequences, uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function random(), which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the random.Random class. You can instantiate your own instances of Random to get generators that don't share state. This is especially useful for multi-threaded programs, creating a different instance of Random for each thread, and using the jumpahead() method to make it likely that the generated sequences seen by each thread don't overlap.

Class Random can also be subclassed if you want to use a different basic generator of your own devising: in that case, override the random(), seed(), getstate(), setstate() and jumpahead() methods. Optionally, a new generator can supply a getrandbits() method — this allows randrange() to produce selections over an arbitrarily large range.

New in version 2.4: the getrandbits() method.

As an example of subclassing, the random module provides the WichmannHill class that implements an alternative generator in pure Python. The class provides a backward compatible way to reproduce results from earlier versions of Python, which used the Wichmann-Hill algorithm as the core generator. Note that this Wichmann-Hill generator can no longer be recommended: its period is too short by contemporary standards, and the sequence generated is known to fail some stringent randomness tests. See the references below for a recent variant that repairs these flaws.

Changed in version 2.3: MersenneTwister replaced Wichmann-Hill as the default generator.

The random module also provides the SystemRandom class which uses the system function os.urandom() to generate random numbers from sources provided by the operating system.

> Warning: The pseudo-random generators of this module should not be used for security purposes. Use os.urandom() or SystemRandom if you require a cryptographically secure pseudo-random number generator.

Bookkeeping functions:

random.seed(a=None)
Initialize internal state of the random number generator.

None or no argument seeds from current time or from an operating system specific randomness source if available (see the os.urandom() function for details on availability).

If a is not None or an int or a long, then hash(a) is used instead. Note that the hash values for some types are nondeterministic when PYTHONHASHSEED is enabled.

Changed in version 2.4: formerly, operating system resources were not used.

random.getstate()
Return an object capturing the current internal state of the generator. This object can be passed to setstate() to restore the state.

New in version 2.1.

Changed in version 2.6: State values produced in Python 2.6 cannot be loaded into earlier versions.

random.setstate(state)
state should have been obtained from a previous call to getstate(), and setstate() restores the internal state of the generator to what it was at the time getstate() was called.

New in version 2.1.

random.jumpahead(n)
Change the internal state to one different from and likely far away from the current state. n is a non-negative integer which is used to scramble the current state vector. This is most useful in multi-threaded programs, in conjunction with multiple instances of the Random class: setstate() or seed() can be used to force all instances into the same internal state, and then jumpahead() can be used to force the instances' states far apart.

New in version 2.1.

Changed in version 2.3: Instead of jumping to a specific state, n steps ahead, jumpahead(n) jumps to another state likely to be separated by many steps.

random.getrandbits(k)
Returns a python long int with k random bits. This method is supplied with the MersenneTwister generator and some other generators may also provide it as an optional part of the API. When available, getrandbits() enables randrange() to handle arbitrarily large ranges.

New in version 2.4.

Functions for integers:

random.randrange(stop)
random.randrange(start, stop[, step])
Return a randomly selected element from range(start, stop, step). This is equivalent to choice(range(start, stop, step)), but doesn't actually build a range object.

New in version 1.5.2.

random.randint(a, b)
Return a random integer N such that a <= N <= b.

Functions for sequences:

random.choice(seq)
Return a random element from the non-empty sequence seq. If seq is empty, raises IndexError.

random.shuffle(x[, random])
: Shuffle the sequence x in place. The optional argument random is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function random().

  Note that for even rather small len(x), the total number of permutations of x is larger than the period of most random number generators; this implies that most permutations of a long sequence can never be generated.

random.sample(population, k)
: Return a k length list of unique elements chosen from the population sequence. Used for random sampling without replacement.

  New in version 2.3.

  Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

  Members of the population need not be hashable or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

  To choose a sample from a range of integers, use an xrange() object as an argument. This is especially fast and space efficient for sampling from a large population: sample(xrange(10000000), 60).

The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distribution's equation, as used in common mathematical practice; most of these equations can be found in any statistics text.

random.random()
: Return the next random floating point number in the range [0.0, 1.0).

random.uniform(a, b)
: Return a random floating point number N such that a <= N <= b for a <= b and b <= N <= a for b < a.

  The end-point value b may or may not be included in the range depending on floating-point rounding in the equation a + (b-a) * random().

random.triangular(low, high, mode)
: Return a random floating point number N such that low <= N <= high and with the specified mode between those bounds. The low and high bounds default to zero and one. The mode argument defaults to the midpoint between the bounds, giving a symmetric distribution.

  New in version 2.6.

random.betavariate(alpha, beta)
: Beta distribution. Conditions on the parameters are alpha > 0 and beta > 0. Returned values range between 0 and 1.

random.expovariate(lambd)
: Exponential distribution. lambd is 1.0 divided by the desired mean. It should be nonzero. (The parameter would be called "lambda", but that is a reserved word in Python.) Returned values range from 0 to positive infinity if lambd is positive, and from negative infinity to 0 if lambd is negative.

random.gammavariate(alpha, beta)
: Gamma distribution. (Not the gamma function!) Conditions on the parameters are alpha > 0 and beta > 0.

  The probability distribution function is:

```
        x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
          math.gamma(alpha) * beta ** alpha
```

random.gauss(mu, sigma)
: Gaussian distribution. mu is the mean, and sigma is the standard deviation. This is slightly faster than the normalvariate() function defined below.

random.lognormvariate(mu, sigma)
: Log normal distribution. If you take the natural logarithm of this distribution, you'll get a normal distribution with mean mu and standard deviation sigma. mu can have any value, and sigma must be greater than zero.

random.normalvariate(mu, sigma)
: Normal distribution. mu is the mean, and sigma is the standard deviation.

random.vonmisesvariate(mu, kappa)
: mu is the mean angle, expressed in radians between 0 and 2*pi, and kappa is the concentration parameter, which must be greater than or equal to zero. If kappa is equal to zero, this distribution reduces to a uniform random angle over the range 0 to 2*pi.

random.paretovariate(alpha)
: Pareto distribution. alpha is the shape parameter.

random.weibullvariate(alpha, beta)
: Weibull distribution. alpha is the scale parameter and beta is the shape parameter.

Alternative Generators:

class random.WichmannHill([seed])
: Class that implements the Wichmann-Hill algorithm as the core generator. Has all of the same methods as Random plus the whseed() method described below. Because this class is implemented in pure Python, it is not threadsafe and may require locks between calls. The period of the generator is 6,953,607,871,644 which is small enough to require care that two independent random sequences do not overlap.

random.whseed([x])
: This is obsolete, supplied for bit-level compatibility with versions of Python prior to 2.1. See seed() for details. whseed() does not guarantee that distinct integer arguments yield distinct internal states, and can yield no more than about 2**24 distinct internal states in all.

class random.SystemRandom([seed])
: Class that uses the os.urandom() function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on software state and sequences are not reproducible. Accordingly, the seed() and jumpahead() methods have no effect and are ignored. The getstate() and setstate() methods raise NotImplementedError if called.

  New in version 2.4.

Examples of basic usage:

```
>>> random.random()        # Random float x, 0.0 <= x < 1.0
0.37444887175646646
>>> random.uniform(1, 10)  # Random float x, 1.0 <= x < 10.0
1.1800146073117523
>>> random.randint(1, 10)  # Integer from 1 to 10, endpoints included
7
>>> random.randrange(0, 101, 2)  # Even integer from 0 to 100
26
```

(continues on next page)

(continued from previous page)

```
>>> random.choice('abcdefghij')  # Choose a random element
'c'

>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)
>>> items
[7, 3, 2, 5, 6, 4, 1]

>>> random.sample([1, 2, 3, 4, 5],  3)  # Choose 3 elements
[4, 1, 5]
```

See also:

M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.

Wichmann, B. A. & Hill, I. D., "Algorithm AS 183: An efficient and portable pseudo-random number generator", Applied Statistics 31 (1982) 188-190.

Complementary-Multiply-with-Carry recipe for a compatible alternative random number generator with a long period and comparatively simple update operations.

9.7 itertools — Functions creating iterators for efficient looping

New in version 2.3.

This module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an "iterator algebra" making it possible to construct specialized tools succinctly and efficiently in pure Python.

For instance, SML provides a tabulation tool: tabulate(f) which produces a sequence f(0), f(1), ... The same effect can be achieved in Python by combining imap() and count() to form imap(f, count()).

These tools and their built-in counterparts also work well with the high-speed functions in the operator module. For example, the multiplication operator can be mapped across two vectors to form an efficient dot-product: sum(imap(operator.mul, vector1, vector2)).

Infinite Iterators:

Iterator	Arguments	Results	Example
count()	start, [step]	start, start+step, start+2*step, ...	count(10) --> 10 11 12 13 14 ...
cycle()	p	p0, p1, ... plast, p0, p1, ...	cycle('ABCD') --> A B C D A B C D ...
repeat()	elem [,n]	elem, elem, elem, ... endlessly or up to n times	repeat(10, 3) --> 10 10 10

Iterators terminating on the shortest input sequence:

Iterator	Arguments	Results	Example
chain()	p, q, ...	p0, p1, ... plast, q0, q1, ...	chain('ABC', 'DEF') --> A B C D E F
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
dropwhile()	pred, seq	seq[n], seq[n+1], starting when pred fails	dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
groupby()	iterable[, keyfunc]	sub-iterators grouped by value of keyfunc(v)	
ifilter()	pred, seq	elements of seq where pred(elem) is true	ifilter(lambda x: x%2, range(10)) --> 1 3 5 7 9
ifilterfalse()	pred, seq	elements of seq where pred(elem) is false	ifilterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
islice()	seq, [start,] stop [, step]	elements from seq[start:stop:step]	islice('ABCDEFG', 2, None) --> C D E F G
imap()	func, p, q, ...	func(p0, q0), func(p1, q1), ...	imap(pow, (2,3,10), (5,2,3)) --> 32 9 1000
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ...	starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
tee()	it, n	it1, it2, ... itn splits one iterator into n	
takewhile()	pred, seq	seq[0], seq[1], until pred fails	takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
izip()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	izip('ABCD', 'xy') --> Ax By
izip_longest()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	izip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-

Combinatoric generators:

Iterator	Arguments	Results
product()	p, q, ... [repeat=1]	cartesian product, equivalent to a nested for-loop
permutations()	p[, r]	r-length tuples, all possible orderings, no repeated elements
combinations()	p, r	r-length tuples, in sorted order, no repeated elements
combinations_with_replacement()	p, r	r-length tuples, in sorted order, with repeated elements
product('ABCD', repeat=2)		AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
permutations('ABCD', 2)		AB AC AD BA BC BD CA CB CD DA DB DC
combinations('ABCD', 2)		AB AC AD BC BD CD
combinations_with_replacement('ABCD', 2)		AA AB AC AD BB BC BD CC CD DD

9.7.1 Itertool functions

The following module functions all construct and return iterators. Some provide streams of infinite length, so they should only be accessed by functions or loops that truncate the stream.

itertools.chain(*iterables)

Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a

single sequence. Roughly equivalent to:

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

classmethod chain.from_iterable(iterable)

Alternate constructor for chain(). Gets chained inputs from a single iterable argument that is evaluated lazily. Roughly equivalent to:

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

New in version 2.6.

itertools.combinations(iterable, r)

Return r length subsequences of elements from the input iterable.

Combinations are emitted in lexicographic sort order. So, if the input iterable is sorted, the combination tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each combination.

Roughly equivalent to:

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = range(r)
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

The code for combinations() can be also expressed as a subsequence of permutations() after filtering entries where the elements are not in sorted order (according to their position in the input pool):

```
def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
```

(continues on next page)

(continued from previous page)

```
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

The number of items returned is n! / r! / (n-r)! when 0 <= r <= n or zero when r > n.

New in version 2.6.

itertools.combinations_with_replacement(iterable, r)

Return r length subsequences of elements from the input iterable allowing individual elements to be repeated more than once.

Combinations are emitted in lexicographic sort order. So, if the input iterable is sorted, the combination tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, the generated combinations will also be unique.

Roughly equivalent to:

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

The code for combinations_with_replacement() can be also expressed as a subsequence of product() after filtering entries where the elements are not in sorted order (according to their position in the input pool):

```
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

The number of items returned is (n+r-1)! / r! / (n-1)! when n > 0.

New in version 2.7.

itertools.compress(data, selectors)

Make an iterator that filters elements from data returning only those that have a corresponding element in selectors that evaluates to True. Stops when either the data or selectors iterables has been exhausted. Roughly equivalent to:

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in izip(data, selectors) if s)
```

New in version 2.7.

itertools.count(start=0, step=1)

Make an iterator that returns evenly spaced values starting with n. Often used as an argument to imap() to generate consecutive data points. Also, used with izip() to add sequence numbers. Equivalent to:

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

When counting with floating point numbers, better accuracy can sometimes be achieved by substituting multiplicative code such as: (start + step * i for i in count()).

Changed in version 2.7: added step argument and allowed non-integer arguments.

itertools.cycle(iterable)

Make an iterator returning elements from the iterable and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely. Roughly equivalent to:

```
def cycle(iterable):
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
              yield element
```

Note, this member of the toolkit may require significant auxiliary storage (depending on the length of the iterable).

itertools.dropwhile(predicate, iterable)

Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element. Note, the iterator does not produce any output until the predicate first becomes false, so it may have a lengthy start-up time. Roughly equivalent to:

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

itertools.groupby(iterable[, key])

Make an iterator that returns consecutive keys and groups from the iterable. The key is a function computing a key value for each element. If not specified or is None, key defaults to an identity function and returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function.

The operation of groupby() is similar to the uniq filter in Unix. It generates a break or new group every time the value of the key function changes (which is why it is usually necessary to have sorted the

data using the same key function). That behavior differs from SQL's GROUP BY which aggregates common elements regardless of their input order.

The returned group is itself an iterator that shares the underlying iterable with groupby(). Because the source is shared, when the groupby() object is advanced, the previous group is no longer visible. So, if that data is needed later, it should be stored as a list:

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

groupby() is roughly equivalent to:

```
class groupby(object):
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = object()
    def __iter__(self):
        return self
    def next(self):
        while self.currkey == self.tgtkey:
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
        self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey))
    def _grouper(self, tgtkey):
        while self.currkey == tgtkey:
            yield self.currvalue
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
```

New in version 2.4.

itertools.ifilter(predicate, iterable)

Make an iterator that filters elements from iterable returning only those for which the predicate is True. If predicate is None, return the items that are true. Roughly equivalent to:

```
def ifilter(predicate, iterable):
    # ifilter(lambda x: x%2, range(10)) --> 1 3 5 7 9
    if predicate is None:
        predicate = bool
    for x in iterable:
        if predicate(x):
            yield x
```

itertools.ifilterfalse(predicate, iterable)

Make an iterator that filters elements from iterable returning only those for which the predicate is False. If predicate is None, return the items that are false. Roughly equivalent to:

```
def ifilterfalse(predicate, iterable):
    # ifilterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.imap(function, *iterables)

Make an iterator that computes the function using arguments from each of the iterables. If function is set to None, then imap() returns the arguments as a tuple. Like map() but stops when the shortest iterable is exhausted instead of filling in None for shorter iterables. The reason for the difference is that infinite iterator arguments are typically an error for map() (because the output is fully evaluated) but represent a common and useful way of supplying arguments to imap(). Roughly equivalent to:

```
def imap(function, *iterables):
    # imap(pow, (2,3,10), (5,2,3)) --> 32 9 1000
    iterables = map(iter, iterables)
    while True:
        args = [next(it) for it in iterables]
        if function is None:
            yield tuple(args)
        else:
            yield function(*args)
```

itertools.islice(iterable, stop)

itertools.islice(iterable, start, stop[, step])

Make an iterator that returns selected elements from the iterable. If start is non-zero, then elements from the iterable are skipped until start is reached. Afterward, elements are returned consecutively unless step is set higher than one which results in items being skipped. If stop is None, then iteration continues until the iterator is exhausted, if at all; otherwise, it stops at the specified position. Unlike regular slicing, islice() does not support negative values for start, stop, or step. Can be used to extract related fields from data where the internal structure has been flattened (for example, a multi-line report may list a name field on every third line). Roughly equivalent to:

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or sys.maxint, s.step or 1
    it = iter(xrange(start, stop, step))
    try:
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in izip(xrange(start), iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(it)
    except StopIteration:
```

(continues on next page)

(continued from previous page)

```
        # Consume to *stop*.
        for i, element in izip(xrange(i + 1, stop), iterable):
            pass
```

If start is None, then iteration starts at zero. If step is None, then the step defaults to one.

Changed in version 2.5: accept None values for default start and step.

itertools.izip(*iterables)

Make an iterator that aggregates elements from each of the iterables. Like zip() except that it returns an iterator instead of a list. Used for lock-step iteration over several iterables at a time. Roughly equivalent to:

```
def izip(*iterables):
    # izip('ABCD', 'xy') --> Ax By
    iterators = map(iter, iterables)
    while iterators:
        yield tuple(map(next, iterators))
```

Changed in version 2.4: When no iterables are specified, returns a zero length iterator instead of raising a TypeError exception.

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using izip(*[iter(s)]*n).

izip() should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables. If those values are important, use izip_longest() instead.

itertools.izip_longest(*iterables[, fillvalue])

Make an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with fillvalue. Iteration continues until the longest iterable is exhausted. Roughly equivalent to:

```
class ZipExhausted(Exception):
    pass

def izip_longest(*args, **kwds):
    # izip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    fillvalue = kwds.get('fillvalue')
    counter = [len(args) - 1]
    def sentinel():
        if not counter[0]:
            raise ZipExhausted
        counter[0] -= 1
        yield fillvalue
    fillers = repeat(fillvalue)
    iterators = [chain(it, sentinel(), fillers) for it in args]
    try:
        while iterators:
            yield tuple(map(next, iterators))
    except ZipExhausted:
        pass
```

If one of the iterables is potentially infinite, then the izip_longest() function should be wrapped with something that limits the number of calls (for example islice() or takewhile()). If not specified, fillvalue defaults to None.

New in version 2.6.

itertools.permutations(iterable[, r])

Return successive r length permutations of elements in the iterable.

If r is not specified or is None, then r defaults to the length of the iterable and all possible full-length permutations are generated.

Permutations are emitted in lexicographic sort order. So, if the input iterable is sorted, the permutation tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each permutation.

Roughly equivalent to:

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = range(n)
    cycles = range(n, n-r, -1)
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

The code for permutations() can be also expressed as a subsequence of product(), filtered to exclude entries with repeated elements (those from the same position in the input pool):

```
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)
```

The number of items returned is n! / (n-r)! when 0 <= r <= n or zero when r > n.

New in version 2.6.

itertools.product(*iterables[, repeat])

Cartesian product of input iterables.

Roughly equivalent to nested for-loops in a generator expression. For example, product(A, B) returns the same as ((x,y) for x in A for y in B).

The nested loops cycle like an odometer with the rightmost element advancing on every iteration. This pattern creates a lexicographic ordering so that if the input's iterables are sorted, the product tuples are emitted in sorted order.

To compute the product of an iterable with itself, specify the number of repetitions with the optional repeat keyword argument. For example, product(A, repeat=4) means the same as product(A, A, A, A).

This function is roughly equivalent to the following code, except that the actual implementation does not build up intermediate results in memory:

```
def product(*args, **kwds):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = map(tuple, args) * kwds.get('repeat', 1)
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

New in version 2.6.

itertools.repeat(object[, times])

Make an iterator that returns object over and over again. Runs indefinitely unless the times argument is specified. Used as argument to imap() for invariant function parameters. Also used with izip() to create constant fields in a tuple record. Roughly equivalent to:

```
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in xrange(times):
            yield object
```

A common use for repeat is to supply a stream of constant values to imap or zip:

```
>>> list(imap(pow, xrange(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

itertools.starmap(function, iterable)

Make an iterator that computes the function using arguments obtained from the iterable. Used instead of imap() when argument parameters are already grouped in tuples from a single iterable (the data has been "pre-zipped"). The difference between imap() and starmap() parallels the distinction between function(a,b) and function(*c). Roughly equivalent to:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

Changed in version 2.6: Previously, starmap() required the function arguments to be tuples. Now, any iterable is allowed.

itertools.takewhile(predicate, iterable)

Make an iterator that returns elements from the iterable as long as the predicate is true. Roughly equivalent to:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

itertools.tee(iterable[, n=2])

Return n independent iterators from a single iterable. Roughly equivalent to:

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:             # when the local deque is empty
                newval = next(it)       # fetch a new value and
                for d in deques:        # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

Once tee() has made a split, the original iterable should not be used anywhere else; otherwise, the iterable could get advanced without the tee objects being informed.

This itertool may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use list() instead of tee().

New in version 2.4.

9.7.2 Recipes

This section shows recipes for creating an extended toolset using the existing itertools as building blocks.

The extended tools offer the same high performance as the underlying toolset. The superior memory performance is kept by processing elements one at a time rather than bringing the whole iterable into memory all at once. Code volume is kept small by linking the tools together in a functional style which helps eliminate temporary variables. High speed is retained by preferring "vectorized" building blocks over the use of for-loops and generators which incur interpreter overhead.

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return imap(function, count(start))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        # feed the entire iterator into a zero-length deque
        collections.deque(iterator, maxlen=0)
```

(continues on next page)

(continued from previous page)

```
    else:
        # advance to the empty slice starting at position n
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)

def all_equal(iterable):
    "Returns True if all the elements are equal to each other"
    g = groupby(iterable)
    return next(g, True) and not next(g, False)

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(imap(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.

    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(imap(operator.mul, vec1, vec2))

def flatten(listOfLists):
    "Flatten one level of nesting"
    return chain.from_iterable(listOfLists)

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.

    Example:  repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return izip(a, b)

def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx
    args = [iter(iterable)] * n
    return izip_longest(fillvalue=fillvalue, *args)
```

(continues on next page)

(continued from previous page)

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    pending = len(iterables)
    nexts = cycle(iter(it).next for it in iterables)
    while pending:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            pending -= 1
            nexts = cycle(islice(nexts, pending))

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    # unique_everseen('ABBCcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in ifilterfalse(seen.__contains__, iterable):
            seen_add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
    # unique_justseen('ABBCcAD', str.lower) --> A B C A D
    return imap(next, imap(itemgetter(1), groupby(iterable, key)))

def iter_except(func, exception, first=None):
    """ Call a function repeatedly until an exception is raised.

    Converts a call-until-exception interface to an iterator interface.
    Like __builtin__.iter(func, sentinel) but uses an exception instead
    of a sentinel to end the loop.

    Examples:
        bsddbiter = iter_except(db.next, bsddb.error, db.first)
        heapiter = iter_except(functools.partial(heappop, h), IndexError)
        dictiter = iter_except(d.popitem, KeyError)
        dequeiter = iter_except(d.popleft, IndexError)
        queueiter = iter_except(q.get_nowait, Queue.Empty)
        setiter = iter_except(s.pop, KeyError)
```

(continues on next page)

(continued from previous page)

```
    """
    try:
        if first is not None:
            yield first()
        while 1:
            yield func()
    except exception:
        pass

def random_product(*args, **kwds):
    "Random selection from itertools.product(*args, **kwds)"
    pools = map(tuple, args) * kwds.get('repeat', 1)
    return tuple(random.choice(pool) for pool in pools)

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(xrange(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Random selection from itertools.combinations_with_replacement(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in xrange(r))
    return tuple(pool[i] for i in indices)

def tee_lookahead(t, i):
    """Inspect the i-th upcomping value from a tee object
       while leaving the tee object at its current position.

       Raise an IndexError if the underlying iterator doesn't
       have enough values.

    """
    for value in islice(t.__copy__(), i, None):
        return value
    raise IndexError(i)
```

Note, many of the above recipes can be optimized by replacing global lookups with local variables defined as default values. For example, the dotproduct recipe can be written as:

```
def dotproduct(vec1, vec2, sum=sum, imap=imap, mul=operator.mul):
    return sum(imap(mul, vec1, vec2))
```

9.8 functools — Higher-order functions and operations on callable objects

New in version 2.5.

Source code: Lib/functools.py

The functools module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.

The functools module defines the following functions:

functools.cmp_to_key(func)
: Transform an old-style comparison function to a key function. Used with tools that accept key functions (such as sorted(), min(), max(), heapq.nlargest(), heapq.nsmallest(), itertools.groupby()). This function is primarily used as a transition tool for programs being converted to Python 3 where comparison functions are no longer supported.

  A comparison function is any callable that accept two arguments, compares them, and returns a negative number for less-than, zero for equality, or a positive number for greater-than. A key function is a callable that accepts one argument and returns another value to be used as the sort key.

  Example:

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-aware sort order
```

  For sorting examples and a brief sorting tutorial, see sortinghowto.

  New in version 2.7.

functools.total_ordering(cls)
: Given a class defining one or more rich comparison ordering methods, this class decorator supplies the rest. This simplifies the effort involved in specifying all of the possible rich comparison operations:

  The class must define one of __lt__(), __le__(), __gt__(), or __ge__(). In addition, the class should supply an __eq__() method.

  For example:

```
@total_ordering
class Student:
    def __eq__(self, other):
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

  New in version 2.7.

functools.reduce(function, iterable[, initializer])
: This is the same function as reduce(). It is made available in this module to allow writing code more forward-compatible with Python 3.

  New in version 2.6.

functools.partial(func[,*args][, **keywords])
: Return a new partial object which when called will behave like func called with the positional arguments args and keyword arguments keywords. If more arguments are supplied to the call, they are appended to args. If additional keyword arguments are supplied, they extend and override keywords. Roughly equivalent to:

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

The partial() is used for partial function application which "freezes" some portion of a function's arguments and/or keywords resulting in a new object with a simplified signature. For example, partial() can be used to create a callable that behaves like the int() function where the base argument defaults to two:

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

functools.update_wrapper(wrapper, wrapped[, assigned][, updated])

Update a wrapper function to look like the wrapped function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants WRAPPER_ASSIGNMENTS (which assigns to the wrapper function's __name__, __module__ and __doc__, the documentation string) and WRAPPER_UPDATES (which updates the wrapper function's __dict__, i.e. the instance dictionary).

The main intended use for this function is in decorator functions which wrap the decorated function and return the wrapper. If the wrapper function is not updated, the metadata of the returned function will reflect the wrapper definition rather than the original function definition, which is typically less than helpful.

functools.wraps(wrapped[, assigned][, updated])

This is a convenience function for invoking update_wrapper() as a function decorator when defining a wrapper function. It is equivalent to partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated). For example:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print 'Calling decorated function'
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print 'Called example function'
...
>>> example()
Calling decorated function
Called example function
```

(continues on next page)

(continued from previous page)

```
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

Without the use of this decorator factory, the name of the example function would have been `'wrapper'`, and the docstring of the original example() would have been lost.

9.8.1 partial Objects

partial objects are callable objects created by partial(). They have three read-only attributes:

partial.func
: A callable object or function. Calls to the partial object will be forwarded to func with new arguments and keywords.

partial.args
: The leftmost positional arguments that will be prepended to the positional arguments provided to a partial object call.

partial.keywords
: The keyword arguments that will be supplied when the partial object is called.

partial objects are like function objects in that they are callable, weak referencable, and can have attributes. There are some important differences. For instance, the __name__ and __doc__ attributes are not created automatically. Also, partial objects defined in classes behave like static methods and do not transform into bound methods during instance attribute look-up.

9.9 operator — Standard operators as functions

The operator module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, operator.add(x, y) is equivalent to the expression x+y. The function names are those used for special class methods; variants without leading and trailing __ are also provided for convenience.

The functions fall into categories that perform object comparisons, logical operations, mathematical operations, sequence operations, and abstract type tests.

The object comparison functions are useful for all objects, and are named after the rich comparison operators they support:

operator.lt(a, b)
operator.le(a, b)
operator.eq(a, b)
operator.ne(a, b)
operator.ge(a, b)
operator.gt(a, b)
operator.__lt__(a, b)
operator.__le__(a, b)
operator.__eq__(a, b)
operator.__ne__(a, b)
operator.__ge__(a, b)
operator.__gt__(a, b)
: Perform “rich comparisons” between a and b. Specifically, lt(a, b) is equivalent to a < b, le(a, b) is equivalent to a <= b, eq(a, b) is equivalent to a == b, ne(a, b) is equivalent to a != b, gt(a, b)

is equivalent to a > b and ge(a, b) is equivalent to a >= b. Note that unlike the built-in cmp(), these functions can return any value, which may or may not be interpretable as a Boolean value. See comparisons for more information about rich comparisons.

New in version 2.2.

The logical operations are also generally applicable to all objects, and support truth tests, identity tests, and boolean operations:

operator.not_(obj)
operator.__not__(obj)
Return the outcome of not obj. (Note that there is no __not__() method for object instances; only the interpreter core defines this operation. The result is affected by the __nonzero__() and __len__() methods.)

operator.truth(obj)
Return True if obj is true, and False otherwise. This is equivalent to using the bool constructor.

operator.is_(a, b)
Return a is b. Tests object identity.

New in version 2.3.

operator.is_not(a, b)
Return a is not b. Tests object identity.

New in version 2.3.

The mathematical and bitwise operations are the most numerous:

operator.abs(obj)
operator.__abs__(obj)
Return the absolute value of obj.

operator.add(a, b)
operator.__add__(a, b)
Return a + b, for a and b numbers.

operator.and_(a, b)
operator.__and__(a, b)
Return the bitwise and of a and b.

operator.div(a, b)
operator.__div__(a, b)
Return a / b when __future__.division is not in effect. This is also known as "classic" division.

operator.floordiv(a, b)
operator.__floordiv__(a, b)
Return a // b.

New in version 2.2.

operator.index(a)
operator.__index__(a)
Return a converted to an integer. Equivalent to a.__index__().

New in version 2.5.

operator.inv(obj)
operator.invert(obj)
operator.__inv__(obj)

operator.__invert__(obj)
Return the bitwise inverse of the number obj. This is equivalent to ~obj.

New in version 2.0: The names invert() and __invert__().

operator.lshift(a, b)
operator.__lshift__(a, b)
Return a shifted left by b.

operator.mod(a, b)
operator.__mod__(a, b)
Return a % b.

operator.mul(a, b)
operator.__mul__(a, b)
Return a * b, for a and b numbers.

operator.neg(obj)
operator.__neg__(obj)
Return obj negated (-obj).

operator.or_(a, b)
operator.__or__(a, b)
Return the bitwise or of a and b.

operator.pos(obj)
operator.__pos__(obj)
Return obj positive (+obj).

operator.pow(a, b)
operator.__pow__(a, b)
Return a ** b, for a and b numbers.

New in version 2.3.

operator.rshift(a, b)
operator.__rshift__(a, b)
Return a shifted right by b.

operator.sub(a, b)
operator.__sub__(a, b)
Return a - b.

operator.truediv(a, b)
operator.__truediv__(a, b)
Return a / b when __future__.division is in effect. This is also known as "true" division.

New in version 2.2.

operator.xor(a, b)
operator.__xor__(a, b)
Return the bitwise exclusive or of a and b.

Operations which work with sequences (some of them with mappings too) include:

operator.concat(a, b)
operator.__concat__(a, b)
Return a + b for a and b sequences.

operator.contains(a, b)
operator.__contains__(a, b)
Return the outcome of the test b in a. Note the reversed operands.

New in version 2.0: The name __contains__().

operator.countOf(a, b)
Return the number of occurrences of b in a.

operator.delitem(a, b)
operator.__delitem__(a, b)
Remove the value of a at index b.

operator.delslice(a, b, c)
operator.__delslice__(a, b, c)
Delete the slice of a from index b to index c-1.

Deprecated since version 2.6: This function is removed in Python 3.x. Use delitem() with a slice index.

operator.getitem(a, b)
operator.__getitem__(a, b)
Return the value of a at index b.

operator.getslice(a, b, c)
operator.__getslice__(a, b, c)
Return the slice of a from index b to index c-1.

Deprecated since version 2.6: This function is removed in Python 3.x. Use getitem() with a slice index.

operator.indexOf(a, b)
Return the index of the first of occurrence of b in a.

operator.repeat(a, b)
operator.__repeat__(a, b)
Deprecated since version 2.7: Use __mul__() instead.

Return a * b where a is a sequence and b is an integer.

operator.sequenceIncludes(...)
Deprecated since version 2.0: Use contains() instead.

Alias for contains().

operator.setitem(a, b, c)
operator.__setitem__(a, b, c)
Set the value of a at index b to c.

operator.setslice(a, b, c, v)
operator.__setslice__(a, b, c, v)
Set the slice of a from index b to index c-1 to the sequence v.

Deprecated since version 2.6: This function is removed in Python 3.x. Use setitem() with a slice index.

Example use of operator functions:

```
>>> # Elementwise multiplication
>>> map(mul, [0, 1, 2, 3], [10, 20, 30, 40])
[0, 20, 60, 120]

>>> # Dot product
>>> sum(map(mul, [0, 1, 2, 3], [10, 20, 30, 40]))
200
```

Many operations have an "in-place" version. The following functions provide a more primitive access to in-place operators than the usual syntax does; for example, the statement x += y is equivalent to x = operator.iadd(x, y). Another way to put it is to say that z = operator.iadd(x, y) is equivalent to the compound statement z = x; z += y.

operator.iadd(a, b)
operator.__iadd__(a, b)
a = iadd(a, b) is equivalent to a += b.

New in version 2.5.

operator.iand(a, b)
operator.__iand__(a, b)
a = iand(a, b) is equivalent to a &= b.

New in version 2.5.

operator.iconcat(a, b)
operator.__iconcat__(a, b)
a = iconcat(a, b) is equivalent to a += b for a and b sequences.

New in version 2.5.

operator.idiv(a, b)
operator.__idiv__(a, b)
a = idiv(a, b) is equivalent to a /= b when __future__.division is not in effect.

New in version 2.5.

operator.ifloordiv(a, b)
operator.__ifloordiv__(a, b)
a = ifloordiv(a, b) is equivalent to a //= b.

New in version 2.5.

operator.ilshift(a, b)
operator.__ilshift__(a, b)
a = ilshift(a, b) is equivalent to a <<= b.

New in version 2.5.

operator.imod(a, b)
operator.__imod__(a, b)
a = imod(a, b) is equivalent to a %= b.

New in version 2.5.

operator.imul(a, b)
operator.__imul__(a, b)
a = imul(a, b) is equivalent to a *= b.

New in version 2.5.

operator.ior(a, b)
operator.__ior__(a, b)
a = ior(a, b) is equivalent to a |= b.

New in version 2.5.

operator.ipow(a, b)
operator.__ipow__(a, b)
a = ipow(a, b) is equivalent to a **= b.

New in version 2.5.

operator.irepeat(a, b)
operator.__irepeat__(a, b)
Deprecated since version 2.7: Use __imul__() instead.

a = irepeat(a, b) is equivalent to a *= b where a is a sequence and b is an integer.

New in version 2.5.

operator.irshift(a, b)
operator.__irshift__(a, b)
a = irshift(a, b) is equivalent to a >>= b.

New in version 2.5.

operator.isub(a, b)
operator.__isub__(a, b)
a = isub(a, b) is equivalent to a -= b.

New in version 2.5.

operator.itruediv(a, b)
operator.__itruediv__(a, b)
a = itruediv(a, b) is equivalent to a /= b when __future__.division is in effect.

New in version 2.5.

operator.ixor(a, b)
operator.__ixor__(a, b)
a = ixor(a, b) is equivalent to a ^= b.

New in version 2.5.

The operator module also defines a few predicates to test the type of objects; however, these are not all reliable. It is preferable to test abstract base classes instead (see collections and numbers for details).

operator.isCallable(obj)
Deprecated since version 2.0: Use isinstance(x, collections.Callable) instead.

Returns true if the object obj can be called like a function, otherwise it returns false. True is returned for functions, bound and unbound methods, class objects, and instance objects which support the __call__() method.

operator.isMappingType(obj)
Deprecated since version 2.7: Use isinstance(x, collections.Mapping) instead.

Returns true if the object obj supports the mapping interface. This is true for dictionaries and all instance objects defining __getitem__().

operator.isNumberType(obj)
Deprecated since version 2.7: Use isinstance(x, numbers.Number) instead.

Returns true if the object obj represents a number. This is true for all numeric types implemented in C.

operator.isSequenceType(obj)
Deprecated since version 2.7: Use isinstance(x, collections.Sequence) instead.

Returns true if the object obj supports the sequence protocol. This returns true for all objects which define sequence methods in C, and for all instance objects defining __getitem__().

The operator module also defines tools for generalized attribute and item lookups. These are useful for making fast field extractors as arguments for map(), sorted(), itertools.groupby(), or other functions that expect a function argument.

operator.attrgetter(attr)
operator.attrgetter(*attrs)
Return a callable object that fetches attr from its operand. If more than one attribute is requested, returns a tuple of attributes. The attribute names can also contain dots. For example:

- After f = attrgetter('name'), the call f(b) returns b.name.
- After f = attrgetter('name', 'date'), the call f(b) returns (b.name, b.date).
- After f = attrgetter('name.first', 'name.last'), the call f(b) returns (b.name.first, b.name.last).

Equivalent to:

```
def attrgetter(*items):
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

New in version 2.4.

Changed in version 2.5: Added support for multiple attributes.

Changed in version 2.6: Added support for dotted attributes.

operator.itemgetter(item)
operator.itemgetter(*items)

Return a callable object that fetches item from its operand using the operand's __getitem__() method. If multiple items are specified, returns a tuple of lookup values. For example:

- After f = itemgetter(2), the call f(r) returns r[2].
- After g = itemgetter(2, 5, 3), the call g(r) returns (r[2], r[5], r[3]).

Equivalent to:

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

The items can be any type accepted by the operand's __getitem__() method. Dictionaries accept any hashable value. Lists, tuples, and strings accept an index or a slice:

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1,3,5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2,None))('ABCDEFG')
'CDEFG'
```

New in version 2.4.

Changed in version 2.5: Added support for multiple item extraction.

Example of using itemgetter() to retrieve specific fields from a tuple record:

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> map(getcount, inventory)
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.methodcaller(name[, args...])

Return a callable object that calls the method name on its operand. If additional arguments and/or keyword arguments are given, they will be given to the method as well. For example:

- After f = methodcaller('name'), the call f(b) returns b.name().
- After f = methodcaller('name', 'foo', bar=1), the call f(b) returns b.name('foo', bar=1).

Equivalent to:

```
def methodcaller(name, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

New in version 2.6.

9.9.1 Mapping Operators to Functions

This table shows how abstract operations correspond to operator symbols in the Python syntax and the functions in the operator module.

Operation	Syntax	Function
Addition	a + b	add(a, b)
Concatenation	seq1 + seq2	concat(seq1, seq2)
Containment Test	obj in seq	contains(seq, obj)
Division	a / b	div(a, b) (without __future__.division)
Division	a / b	truediv(a, b) (with __future__.division)
Division	a // b	floordiv(a, b)
Bitwise And	a & b	and_(a, b)
Bitwise Exclusive Or	a ^ b	xor(a, b)
Bitwise Inversion	~ a	invert(a)
Bitwise Or	a | b	or_(a, b)
Exponentiation	a ** b	pow(a, b)
Identity	a is b	is_(a, b)
Identity	a is not b	is_not(a, b)
Indexed Assignment	obj[k] = v	setitem(obj, k, v)
Indexed Deletion	del obj[k]	delitem(obj, k)
Indexing	obj[k]	getitem(obj, k)
Left Shift	a << b	lshift(a, b)
Modulo	a % b	mod(a, b)
Multiplication	a * b	mul(a, b)
Negation (Arithmetic)	- a	neg(a)

Continued on next page

Table 1 – continued from previous page

Operation	Syntax	Function
Negation (Logical)	not a	not_(a)
Positive	+ a	pos(a)
Right Shift	a >> b	rshift(a, b)
Sequence Repetition	seq * i	repeat(seq, i)
Slice Assignment	seq[i:j] = values	setitem(seq, slice(i, j), values)
Slice Deletion	del seq[i:j]	delitem(seq, slice(i, j))
Slicing	seq[i:j]	getitem(seq, slice(i, j))
String Formatting	s % obj	mod(s, obj)
Subtraction	a - b	sub(a, b)
Truth Test	obj	truth(obj)
Ordering	a < b	lt(a, b)
Ordering	a <= b	le(a, b)
Equality	a == b	eq(a, b)
Difference	a != b	ne(a, b)
Ordering	a >= b	ge(a, b)
Ordering	a > b	gt(a, b)

CHAPTER

TEN

FILE AND DIRECTORY ACCESS

The modules described in this chapter deal with disk files and directories. For example, there are modules for reading the properties of files, manipulating paths in a portable way, and creating temporary files. The full list of modules in this chapter is:

10.1 os.path — Common pathname manipulations

This module implements some useful functions on pathnames. To read or write files see open(), and for accessing the filesystem see the os module.

Note: On Windows, many of these functions do not properly support UNC pathnames. splitunc() and ismount() do handle them correctly.

Unlike a unix shell, Python does not do any automatic path expansions. Functions such as expanduser() and expandvars() can be invoked explicitly when an application desires shell-like path expansion. (See also the glob module.)

Note: Since different operating systems have different path name conventions, there are several versions of this module in the standard library. The os.path module is always the path module suitable for the operating system Python is running on, and therefore usable for local paths. However, you can also import and use the individual modules if you want to manipulate a path that is always in one of the different formats. They all have the same interface:

- posixpath for UNIX-style paths
- ntpath for Windows paths
- macpath for old-style MacOS paths
- os2emxpath for OS/2 EMX paths

---

os.path.abspath(path)
: Return a normalized absolutized version of the pathname path. On most platforms, this is equivalent to calling the function normpath() as follows: normpath(join(os.getcwd(), path)).

  New in version 1.5.2.

os.path.basename(path)
: Return the base name of pathname path. This is the second element of the pair returned by passing path to the function split(). Note that the result of this function is different from the Unix basename

program; where basename for '/foo/bar/' returns 'bar', the basename() function returns an empty string ('').

os.path.commonprefix(list)
Return the longest path prefix (taken character-by-character) that is a prefix of all paths in list. If list is empty, return the empty string (''). Note that this may return invalid paths because it works a character at a time.

os.path.dirname(path)
Return the directory name of pathname path. This is the first element of the pair returned by passing path to the function split().

os.path.exists(path)
Return True if path refers to an existing path. Returns False for broken symbolic links. On some platforms, this function may return False if permission is not granted to execute os.stat() on the requested file, even if the path physically exists.

os.path.lexists(path)
Return True if path refers to an existing path. Returns True for broken symbolic links. Equivalent to exists() on platforms lacking os.lstat().

New in version 2.4.

os.path.expanduser(path)
On Unix and Windows, return the argument with an initial component of ~ or ~user replaced by that user's home directory.

On Unix, an initial ~ is replaced by the environment variable HOME if it is set; otherwise the current user's home directory is looked up in the password directory through the built-in module pwd. An initial ~user is looked up directly in the password directory.

On Windows, HOME and USERPROFILE will be used if set, otherwise a combination of HOMEPATH and HOMEDRIVE will be used. An initial ~user is handled by stripping the last directory component from the created user path derived above.

If the expansion fails or if the path does not begin with a tilde, the path is returned unchanged.

os.path.expandvars(path)
Return the argument with environment variables expanded. Substrings of the form $name or ${name} are replaced by the value of environment variable name. Malformed variable names and references to non-existing variables are left unchanged.

On Windows, %name% expansions are supported in addition to $name and ${name}.

os.path.getatime(path)
Return the time of last access of path. The return value is a number giving the number of seconds since the epoch (see the time module). Raise os.error if the file does not exist or is inaccessible.

New in version 1.5.2.

Changed in version 2.3: If os.stat_float_times() returns True, the result is a floating point number.

os.path.getmtime(path)
Return the time of last modification of path. The return value is a number giving the number of seconds since the epoch (see the time module). Raise os.error if the file does not exist or is inaccessible.

New in version 1.5.2.

Changed in version 2.3: If os.stat_float_times() returns True, the result is a floating point number.

os.path.getctime(path)
Return the system's ctime which, on some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time for path. The return value is a number giving the

number of seconds since the epoch (see the time module). Raise os.error if the file does not exist or is inaccessible.

New in version 2.3.

os.path.getsize(path)
Return the size, in bytes, of path. Raise os.error if the file does not exist or is inaccessible.

New in version 1.5.2.

os.path.isabs(path)
Return True if path is an absolute pathname. On Unix, that means it begins with a slash, on Windows that it begins with a (back)slash after chopping off a potential drive letter.

os.path.isfile(path)
Return True if path is an existing regular file. This follows symbolic links, so both islink() and isfile() can be true for the same path.

os.path.isdir(path)
Return True if path is an existing directory. This follows symbolic links, so both islink() and isdir() can be true for the same path.

os.path.islink(path)
Return True if path refers to a directory entry that is a symbolic link. Always False if symbolic links are not supported by the Python runtime.

os.path.ismount(path)
Return True if pathname path is a mount point: a point in a file system where a different file system has been mounted. The function checks whether path's parent, path/.., is on a different device than path, or whether path/.. and path point to the same i-node on the same device — this should detect mount points for all Unix and POSIX variants.

os.path.join(path, *paths)
Join one or more path components intelligently. The return value is the concatenation of path and any members of *paths with exactly one directory separator (os.sep) following each non-empty part except the last, meaning that the result will only end in a separator if the last part is empty. If a component is an absolute path, all previous components are thrown away and joining continues from the absolute path component.

On Windows, the drive letter is not reset when an absolute path component (e.g., r'\foo') is encountered. If a component contains a drive letter, all previous components are thrown away and the drive letter is reset. Note that since there is a current directory for each drive, os.path.join("c:", "foo") represents a path relative to the current directory on drive C: (c:foo), not c:\foo.

os.path.normcase(path)
Normalize the case of a pathname. On Unix and Mac OS X, this returns the path unchanged; on case-insensitive filesystems, it converts the path to lowercase. On Windows, it also converts forward slashes to backward slashes.

os.path.normpath(path)
Normalize a pathname by collapsing redundant separators and up-level references so that A//B, A/B/, A/./B and A/foo/../B all become A/B. This string manipulation may change the meaning of a path that contains symbolic links. On Windows, it converts forward slashes to backward slashes. To normalize case, use normcase().

os.path.realpath(path)
Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path (if they are supported by the operating system).

New in version 2.2.

os.path.relpath(path[, start])
Return a relative filepath to path either from the current directory or from an optional start directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of path or start.

start defaults to os.curdir.

Availability: Windows, Unix.

New in version 2.6.

os.path.samefile(path1, path2)
Return True if both pathname arguments refer to the same file or directory (as indicated by device number and i-node number). Raise an exception if an os.stat() call on either pathname fails.

Availability: Unix.

os.path.sameopenfile(fp1, fp2)
Return True if the file descriptors fp1 and fp2 refer to the same file.

Availability: Unix.

os.path.samestat(stat1, stat2)
Return True if the stat tuples stat1 and stat2 refer to the same file. These structures may have been returned by os.fstat(), os.lstat(), or os.stat(). This function implements the underlying comparison used by samefile() and sameopenfile().

Availability: Unix.

os.path.split(path)
Split the pathname path into a pair, (head, tail) where tail is the last pathname component and head is everything leading up to that. The tail part will never contain a slash; if path ends in a slash, tail will be empty. If there is no slash in path, head will be empty. If path is empty, both head and tail are empty. Trailing slashes are stripped from head unless it is the root (one or more slashes only). In all cases, join(head, tail) returns a path to the same location as path (but the strings may differ). Also see the functions dirname() and basename().

os.path.splitdrive(path)
Split the pathname path into a pair (drive, tail) where drive is either a drive specification or the empty string. On systems which do not use drive specifications, drive will always be the empty string. In all cases, drive + tail will be the same as path.

New in version 1.3.

os.path.splitext(path)
Split the pathname path into a pair (root, ext) such that root + ext == path, and ext is empty or begins with a period and contains at most one period. Leading periods on the basename are ignored; splitext('.cshrc') returns ('.cshrc', '').

Changed in version 2.6: Earlier versions could produce an empty root when the only period was the first character.

os.path.splitunc(path)
Split the pathname path into a pair (unc, rest) so that unc is the UNC mount point (such as r'\\host\mount'), if present, and rest the rest of the path (such as r'\path\file.ext'). For paths containing drive letters, unc will always be the empty string.

Availability: Windows.

os.path.walk(path, visit, arg)
Calls the function visit with arguments (arg, dirname, names) for each directory in the directory tree rooted at path (including path itself, if it is a directory). The argument dirname specifies the visited

directory, the argument names lists the files in the directory (gotten from os.listdir(dirname)). The visit function may modify names to influence the set of directories visited below dirname, e.g. to avoid visiting certain parts of the tree. (The object referred to by names must be modified in place, using del or slice assignment.)

Note: Symbolic links to directories are not treated as subdirectories, and that walk() therefore will not visit them. To visit linked directories you must identify them with os.path.islink(file) and os.path.isdir(file), and invoke walk() as necessary.

Note: This function is deprecated and has been removed in Python 3 in favor of os.walk().

---

os.path.supports_unicode_filenames
True if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system).

New in version 2.3.

10.2 fileinput — Iterate over lines from multiple input streams

Source code: Lib/fileinput.py

This module implements a helper class and functions to quickly write a loop over standard input or a list of files. If you just want to read or write one file see open().

The typical use is:

```
import fileinput
for line in fileinput.input():
    process(line)
```

This iterates over the lines of all files listed in sys.argv[1:], defaulting to sys.stdin if the list is empty. If a filename is `'-'`, it is also replaced by sys.stdin. To specify an alternative list of filenames, pass it as the first argument to input(). A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the mode parameter in the call to input() or FileInput(). If an I/O error occurs during opening or reading a file, IOError is raised.

If sys.stdin is used more than once, the second and further use will return no lines, except perhaps for interactive use, or if it has been explicitly reset (e.g. using sys.stdin.seek(0)).

Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all is when the last file opened is empty.

Lines are returned with any newlines intact, which means that the last line in a file may not have one.

You can control how files are opened by providing an opening hook via the openhook parameter to fileinput.input() or FileInput(). The hook must be a function that takes two arguments, filename and mode, and returns an accordingly opened file-like object. Two useful hooks are already provided by this module.

The following function is the primary interface of this module:

fileinput.input([files[, inplace[, backup[, bufsize[, mode[, openhook]]]]]])
Create an instance of the FileInput class. The instance will be used as global state for the functions

of this module, and is also returned to use during iteration. The parameters to this function will be passed along to the constructor of the FileInput class.

Changed in version 2.5: Added the mode and openhook parameters.

Changed in version 2.7.12: The bufsize parameter is no longer used.

The following functions use the global state created by fileinput.input(); if there is no active state, RuntimeError is raised.

fileinput.filename()
Return the name of the file currently being read. Before the first line has been read, returns None.

fileinput.fileno()
Return the integer "file descriptor" for the current file. When no file is opened (before the first line and between files), returns -1.

New in version 2.5.

fileinput.lineno()
Return the cumulative line number of the line that has just been read. Before the first line has been read, returns 0. After the last line of the last file has been read, returns the line number of that line.

fileinput.filelineno()
Return the line number in the current file. Before the first line has been read, returns 0. After the last line of the last file has been read, returns the line number of that line within the file.

fileinput.isfirstline()
Returns true if the line just read is the first line of its file, otherwise returns false.

fileinput.isstdin()
Returns true if the last line was read from sys.stdin, otherwise returns false.

fileinput.nextfile()
Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

fileinput.close()
Close the sequence.

The class which implements the sequence behavior provided by the module is available for subclassing as well:

class fileinput.FileInput([files[, inplace[, backup[, bufsize[, mode[, openhook]]]]]])
Class FileInput is the implementation; its methods filename(), fileno(), lineno(), filelineno(), isfirstline(), isstdin(), nextfile() and close() correspond to the functions of the same name in the module. In addition it has a readline() method which returns the next input line, and a __getitem__() method which implements the sequence behavior. The sequence must be accessed in strictly sequential order; random access and readline() cannot be mixed.

With mode you can specify which file mode will be passed to open(). It must be one of 'r', 'rU', 'U' and 'rb'.

The openhook, when given, must be a function that takes two arguments, filename and mode, and returns an accordingly opened file-like object. You cannot use inplace and openhook together.

Changed in version 2.5: Added the mode and openhook parameters.

Changed in version 2.7.12: The bufsize parameter is no longer used.

Optional in-place filtering: if the keyword argument inplace=1 is passed to fileinput.input() or to the FileInput constructor, the file is moved to a backup file and standard output is directed to the input file (if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to write a filter that rewrites its input file in place. If the backup parameter is given (typically as backup='.<some extension>'), it specifies the extension for the backup file, and the backup file remains around; by default, the extension is '.bak' and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

Note: The current implementation does not work for MS-DOS 8+3 filesystems.

The two following opening hooks are provided by this module:

fileinput.hook_compressed(filename, mode)
: Transparently opens files compressed with gzip and bzip2 (recognized by the extensions '.gz' and '.bz2') using the gzip and bz2 modules. If the filename extension is not '.gz' or '.bz2', the file is opened normally (ie, using open() without any decompression).

  Usage example: fi = fileinput.FileInput(openhook=fileinput.hook_compressed)

  New in version 2.5.

fileinput.hook_encoded(encoding)
: Returns a hook which opens each file with io.open(), using the given encoding to read the file.

  Usage example: fi = fileinput.FileInput(openhook=fileinput.hook_encoded("iso-8859-1"))

  Note: With this hook, FileInput might return Unicode strings depending on the specified encoding.

  New in version 2.5.

10.3 stat — Interpreting stat() results

Source code: Lib/stat.py

The stat module defines constants and functions for interpreting the results of os.stat(), os.fstat() and os.lstat() (if they exist). For complete details about the stat(), fstat() and lstat() calls, consult the documentation for your system.

The stat module defines the following functions to test for specific file types:

stat.S_ISDIR(mode)
: Return non-zero if the mode is from a directory.

stat.S_ISCHR(mode)
: Return non-zero if the mode is from a character special device file.

stat.S_ISBLK(mode)
: Return non-zero if the mode is from a block special device file.

stat.S_ISREG(mode)
: Return non-zero if the mode is from a regular file.

stat.S_ISFIFO(mode)
: Return non-zero if the mode is from a FIFO (named pipe).

stat.S_ISLNK(mode)
Return non-zero if the mode is from a symbolic link.

stat.S_ISSOCK(mode)
Return non-zero if the mode is from a socket.

Two additional functions are defined for more general manipulation of the file's mode:

stat.S_IMODE(mode)
Return the portion of the file's mode that can be set by os.chmod()—that is, the file's permission bits, plus the sticky bit, set-group-id, and set-user-id bits (on systems that support them).

stat.S_IFMT(mode)
Return the portion of the file's mode that describes the file type (used by the S_IS*() functions above).

Normally, you would use the os.path.is*() functions for testing the type of a file; the functions here are useful when you are doing multiple tests of the same file and wish to avoid the overhead of the stat() system call for each test. These are also useful when checking for information about a file that isn't handled by os.path, like the tests for block and character devices.

Example:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print 'Skipping %s' % pathname

def visitfile(file):
    print 'visiting', file

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

All the variables below are simply symbolic indexes into the 10-tuple returned by os.stat(), os.fstat() or os.lstat().

stat.ST_MODE
Inode protection mode.

stat.ST_INO
Inode number.

stat.ST_DEV
Device inode resides on.

stat.ST_NLINK
Number of links to the inode.

stat.ST_UID
User id of the owner.

stat.ST_GID
Group id of the owner.

stat.ST_SIZE
Size in bytes of a plain file; amount of data waiting on some special files.

stat.ST_ATIME
Time of last access.

stat.ST_MTIME
Time of last modification.

stat.ST_CTIME
The "ctime" as reported by the operating system. On some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time (see platform documentation for details).

The interpretation of "file size" changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most flavors of Unix (including Linux in particular), the "size" is the number of bytes waiting to be read at the time of the call to os.stat(), os.fstat(), or os.lstat(); this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

The variables below define the flags used in the ST_MODE field.

Use of the functions above is more portable than use of the first set of flags:

stat.S_IFSOCK
Socket.

stat.S_IFLNK
Symbolic link.

stat.S_IFREG
Regular file.

stat.S_IFBLK
Block device.

stat.S_IFDIR
Directory.

stat.S_IFCHR
Character device.

stat.S_IFIFO
FIFO.

The following flags can also be used in the mode argument of os.chmod():

stat.S_ISUID
Set UID bit.

stat.S_ISGID
Set-group-ID bit. This bit has several special uses. For a directory it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the directory, not from

the effective group ID of the creating process, and directories created there will also get the S_ISGID bit set. For a file that does not have the group execution bit (S_IXGRP) set, the set-group-ID bit indicates mandatory file/record locking (see also S_ENFMT).

stat.S_ISVTX
Sticky bit. When this bit is set on a directory it means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, or by a privileged process.

stat.S_IRWXU
Mask for file owner permissions.

stat.S_IRUSR
Owner has read permission.

stat.S_IWUSR
Owner has write permission.

stat.S_IXUSR
Owner has execute permission.

stat.S_IRWXG
Mask for group permissions.

stat.S_IRGRP
Group has read permission.

stat.S_IWGRP
Group has write permission.

stat.S_IXGRP
Group has execute permission.

stat.S_IRWXO
Mask for permissions for others (not in group).

stat.S_IROTH
Others have read permission.

stat.S_IWOTH
Others have write permission.

stat.S_IXOTH
Others have execute permission.

stat.S_ENFMT
System V file locking enforcement. This flag is shared with S_ISGID: file/record locking is enforced on files that do not have the group execution bit (S_IXGRP) set.

stat.S_IREAD
Unix V7 synonym for S_IRUSR.

stat.S_IWRITE
Unix V7 synonym for S_IWUSR.

stat.S_IEXEC
Unix V7 synonym for S_IXUSR.

The following flags can be used in the flags argument of os.chflags():

stat.UF_NODUMP
Do not dump the file.

stat.UF_IMMUTABLE
The file may not be changed.

stat.UF_APPEND
: The file may only be appended to.

stat.UF_OPAQUE
: The directory is opaque when viewed through a union stack.

stat.UF_NOUNLINK
: The file may not be renamed or deleted.

stat.UF_COMPRESSED
: The file is stored compressed (Mac OS X 10.6+).

stat.UF_HIDDEN
: The file should not be displayed in a GUI (Mac OS X 10.5+).

stat.SF_ARCHIVED
: The file may be archived.

stat.SF_IMMUTABLE
: The file may not be changed.

stat.SF_APPEND
: The file may only be appended to.

stat.SF_NOUNLINK
: The file may not be renamed or deleted.

stat.SF_SNAPSHOT
: The file is a snapshot file.

See the *BSD or Mac OS systems man page chflags(2) for more information.

10.4 statvfs — Constants used with os.statvfs()

Deprecated since version 2.6: The statvfs module has been removed in Python 3.

The statvfs module defines constants so interpreting the result if os.statvfs(), which returns a tuple, can be made without remembering "magic numbers." Each of the constants defined in this module is the index of the entry in the tuple returned by os.statvfs() that contains the specified information.

statvfs.F_BSIZE
: Preferred file system block size.

statvfs.F_FRSIZE
: Fundamental file system block size.

statvfs.F_BLOCKS
: Total number of blocks in the filesystem.

statvfs.F_BFREE
: Total number of free blocks.

statvfs.F_BAVAIL
: Free blocks available to non-super user.

statvfs.F_FILES
: Total number of file nodes.

statvfs.F_FFREE
: Total number of free file nodes.

statvfs.F_FAVAIL
: Free nodes available to non-super user.

statvfs.F_FLAG
: Flags. System dependent: see statvfs() man page.

statvfs.F_NAMEMAX
: Maximum file name length.

10.5 filecmp — File and Directory Comparisons

Source code: Lib/filecmp.py

The filecmp module defines functions to compare files and directories, with various optional time/correctness trade-offs. For comparing files, see also the difflib module.

The filecmp module defines the following functions:

filecmp.cmp(f1, f2[, shallow])
: Compare the files named f1 and f2, returning True if they seem equal, False otherwise.

  Unless shallow is given and is false, files with identical os.stat() signatures are taken to be equal.

  Files that were compared using this function will not be compared again unless their os.stat() signature changes.

  Note that no external programs are called from this function, giving it portability and efficiency.

filecmp.cmpfiles(dir1, dir2, common[, shallow])
: Compare the files in the two directories dir1 and dir2 whose names are given by common.

  Returns three lists of file names: match, mismatch, errors. match contains the list of files that match, mismatch contains the names of those that don't, and errors lists the names of files which could not be compared. Files are listed in errors if they don't exist in one of the directories, the user lacks permission to read them or if the comparison could not be done for some other reason.

  The shallow parameter has the same meaning and default value as for filecmp.cmp().

  For example, cmpfiles('a', 'b', ['c', 'd/e']) will compare a/c with b/c and a/d/e with b/d/e. 'c' and 'd/e' will each be in one of the three returned lists.

Example:

```
>>> import filecmp
>>> filecmp.cmp('undoc.rst', 'undoc.rst')
True
>>> filecmp.cmp('undoc.rst', 'index.rst')
False
```

10.5.1 The dircmp class

dircmp instances are built using this constructor:

class filecmp.dircmp(a, b[, ignore[, hide]])
: Construct a new directory comparison object, to compare the directories a and b. ignore is a list of

names to ignore, and defaults to ['RCS', 'CVS', 'tags']. hide is a list of names to hide, and defaults to [os.curdir, os.pardir].

The dircmp class compares files by doing shallow comparisons as described for filecmp.cmp().

The dircmp class provides the following methods:

report()
: Print (to sys.stdout) a comparison between a and b.

report_partial_closure()
: Print a comparison between a and b and common immediate subdirectories.

report_full_closure()
: Print a comparison between a and b and common subdirectories (recursively).

The dircmp class offers a number of interesting attributes that may be used to get various bits of information about the directory trees being compared.

Note that via __getattr__() hooks, all attributes are computed lazily, so there is no speed penalty if only those attributes which are lightweight to compute are used.

left
: The directory a.

right
: The directory b.

left_list
: Files and subdirectories in a, filtered by hide and ignore.

right_list
: Files and subdirectories in b, filtered by hide and ignore.

common
: Files and subdirectories in both a and b.

left_only
: Files and subdirectories only in a.

right_only
: Files and subdirectories only in b.

common_dirs
: Subdirectories in both a and b.

common_files
: Files in both a and b

common_funny
: Names in both a and b, such that the type differs between the directories, or names for which os.stat() reports an error.

same_files
: Files which are identical in both a and b, using the class's file comparison operator.

diff_files
: Files which are in both a and b, whose contents differ according to the class's file comparison operator.

funny_files
: Files which are in both a and b, but could not be compared.

subdirs
: A dictionary mapping names in common_dirs to dircmp objects.

Here is a simplified example of using the subdirs attribute to search recursively through two directories to show common different files:

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print "diff_file %s found in %s and %s" % (name, dcmp.left,
...               dcmp.right)
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

10.6 tempfile — Generate temporary files and directories

Source code: Lib/tempfile.py

This module generates temporary files and directories. It works on all supported platforms.

In version 2.3 of Python, this module was overhauled for enhanced security. It now provides three new functions, NamedTemporaryFile(), mkstemp(), and mkdtemp(), which should eliminate all remaining need to use the insecure mktemp() function. Temporary file names created by this module no longer contain the process ID; instead a string of six random characters is used.

Also, all the user-callable functions now take additional arguments which allow direct control over the location and name of temporary files. It is no longer necessary to use the global tempdir and template variables. To maintain backward compatibility, the argument order is somewhat odd; it is recommended to use keyword arguments for clarity.

The module defines the following user-callable functions:

tempfile.TemporaryFile([mode='w+b'[, bufsize=-1[, suffix=''[, prefix='tmp'[, dir=None]]]]])
: Return a file-like object that can be used as a temporary storage area. The file is created using mkstemp(). It will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected). Under Unix, the directory entry for the file is removed immediately after the file is created. Other platforms do not support this; your code should not rely on a temporary file created using this function having or not having a visible name in the file system.

  The mode parameter defaults to `'w+b'` so that the file created can be read and written without being closed. Binary mode is used so that it behaves consistently on all platforms without regard for the data that is stored. bufsize defaults to -1, meaning that the operating system default is used.

  The dir, prefix and suffix parameters are passed to mkstemp().

  The returned object is a true file object on POSIX platforms. On other platforms, it is a file-like object whose file attribute is the underlying true file object. This file-like object can be used in a with statement, just like a normal file.

tempfile.NamedTemporaryFile([mode='w+b'[, bufsize=-1[, suffix=''[, prefix='tmp'[, dir=None[, delete=True]]]]]])
: This function operates exactly as TemporaryFile() does, except that the file is guaranteed to have

a visible name in the file system (on Unix, the directory entry is not unlinked). That name can be retrieved from the name attribute of the returned file-like object. Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on Unix; it cannot on Windows NT or later). If delete is true (the default), the file is deleted as soon as it is closed.

The returned object is always a file-like object whose file attribute is the underlying true file object. This file-like object can be used in a with statement, just like a normal file.

New in version 2.3.

New in version 2.6: The delete parameter.

tempfile.SpooledTemporaryFile([max_size=0[, mode='w+b'[, bufsize=-1[, suffix=''[, prefix='tmp'[, dir=None]]]]]])

This function operates exactly as TemporaryFile() does, except that data is spooled in memory until the file size exceeds max_size, or until the file's fileno() method is called, at which point the contents are written to disk and operation proceeds as with TemporaryFile(). Also, it's truncate method does not accept a size argument.

The resulting file has one additional method, rollover(), which causes the file to roll over to an on-disk file regardless of its size.

The returned object is a file-like object whose _file attribute is either a StringIO object or a true file object, depending on whether rollover() has been called. This file-like object can be used in a with statement, just like a normal file.

New in version 2.6.

tempfile.mkstemp([suffix=''[, prefix='tmp'[, dir=None[, text=False]]]])

Creates a temporary file in the most secure manner possible. There are no race conditions in the file's creation, assuming that the platform properly implements the os.O_EXCL flag for os.open(). The file is readable and writable only by the creating user ID. If the platform uses permission bits to indicate whether a file is executable, the file is executable by no one. The file descriptor is not inherited by child processes.

Unlike TemporaryFile(), the user of mkstemp() is responsible for deleting the temporary file when done with it.

If suffix is specified, the file name will end with that suffix, otherwise there will be no suffix. mkstemp() does not put a dot between the file name and the suffix; if you need one, put it at the beginning of suffix.

If prefix is specified, the file name will begin with that prefix; otherwise, a default prefix is used.

If dir is specified, the file will be created in that directory; otherwise, a default directory is used. The default directory is chosen from a platform-dependent list, but the user of the application can control the directory location by setting the TMPDIR, TEMP or TMP environment variables. There is thus no guarantee that the generated filename will have any nice properties, such as not requiring quoting when passed to external commands via os.popen().

If text is specified, it indicates whether to open the file in binary mode (the default) or text mode. On some platforms, this makes no difference.

mkstemp() returns a tuple containing an OS-level handle to an open file (as would be returned by os.open()) and the absolute pathname of that file, in that order.

New in version 2.3.

tempfile.mkdtemp([suffix=''[, prefix='tmp'[, dir=None]]])

Creates a temporary directory in the most secure manner possible. There are no race conditions in the

directory's creation. The directory is readable, writable, and searchable only by the creating user ID.

The user of mkdtemp() is responsible for deleting the temporary directory and its contents when done with it.

The prefix, suffix, and dir arguments are the same as for mkstemp().

mkdtemp() returns the absolute pathname of the new directory.

New in version 2.3.

tempfile.mktemp([suffix=''[, prefix='tmp'[, dir=None]]])
Deprecated since version 2.3: Use mkstemp() instead.

Return an absolute pathname of a file that did not exist at the time the call is made. The prefix, suffix, and dir arguments are the same as for mkstemp().

> Warning: Use of this function may introduce a security hole in your program. By the time you get around to doing anything with the file name it returns, someone else may have beaten you to the punch. mktemp() usage can be replaced easily with NamedTemporaryFile(), passing it the delete=False parameter:
>
> ```
> >>> f = NamedTemporaryFile(delete=False)
> >>> f
> <open file '<fdopen>', mode 'w+b' at 0x384698>
> >>> f.name
> '/var/folders/5q/5qTPn6xq2RaWqk+1Ytw3-U+++TI/-Tmp-/tmpG7V1Y0'
> >>> f.write("Hello World!\n")
> >>> f.close()
> >>> os.unlink(f.name)
> >>> os.path.exists(f.name)
> False
> ```

The module uses a global variable that tell it how to construct a temporary name. They are initialized at the first call to any of the functions above. The caller may change them, but this is discouraged; use the appropriate function arguments, instead.

tempfile.tempdir
When set to a value other than None, this variable defines the default value for the dir argument to all the functions defined in this module.

If tempdir is unset or None at any call to any of the above functions, Python searches a standard list of directories and sets tempdir to the first one which the calling user can create files in. The list is:

1. The directory named by the TMPDIR environment variable.
2. The directory named by the TEMP environment variable.
3. The directory named by the TMP environment variable.
4. A platform-specific location:
   - On RiscOS, the directory named by the Wimp$ScrapDir environment variable.
   - On Windows, the directories C:\TEMP, C:\TMP, \TEMP, and \TMP, in that order.
   - On all other platforms, the directories /tmp, /var/tmp, and /usr/tmp, in that order.
5. As a last resort, the current working directory.

tempfile.gettempdir()
Return the directory currently selected to create temporary files in. If tempdir is not None, this simply returns its contents; otherwise, the search described above is performed, and the result returned.

New in version 2.3.

tempfile.template
Deprecated since version 2.0: Use gettempprefix() instead.

When set to a value other than None, this variable defines the prefix of the final component of the filenames returned by mktemp(). A string of six random letters and digits is appended to the prefix to make the filename unique. The default prefix is tmp.

Older versions of this module used to require that template be set to None after a call to os.fork(); this has not been necessary since version 1.5.2.

tempfile.gettempprefix()
Return the filename prefix used to create temporary files. This does not contain the directory component. Using this function is preferred over reading the template variable directly.

New in version 1.5.2.

10.7 glob — Unix style pathname pattern expansion

Source code: Lib/glob.py

The glob module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but *, ?, and character ranges expressed with [] will be correctly matched. This is done by using the os.listdir() and fnmatch.fnmatch() functions in concert, and not by actually invoking a subshell. Note that unlike fnmatch.fnmatch(), glob treats filenames beginning with a dot (.) as special cases. (For tilde and shell variable expansion, use os.path.expanduser() and os.path.expandvars().)

For a literal match, wrap the meta-characters in brackets. For example, '[?]' matches the character '?'.

glob.glob(pathname)
Return a possibly-empty list of path names that match pathname, which must be a string containing a path specification. pathname can be either absolute (like /usr/src/Python-1.5/Makefile) or relative (like ../../Tools/*/*.gif), and can contain shell-style wildcards. Broken symlinks are included in the results (as in the shell).

glob.iglob(pathname)
Return an iterator which yields the same values as glob() without actually storing them all simultaneously.

New in version 2.5.

For example, consider a directory containing only the following files: 1.gif, 2.txt, and card.gif. glob() will produce the following results. Notice how any leading components of the path are preserved.

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

If the directory contains files starting with . they won't be matched by default. For example, consider a directory containing card.gif and .card.gif:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

See also:

Module fnmatch Shell-style filename (not path) expansion

10.8 fnmatch — Unix filename pattern matching

Source code: Lib/fnmatch.py

This module provides support for Unix shell-style wildcards, which are not the same as regular expressions (which are documented in the re module). The special characters used in shell-style wildcards are:

Pattern	Meaning
*	matches everything
?	matches any single character
[seq]	matches any character in seq
[!seq]	matches any character not in seq

For a literal match, wrap the meta-characters in brackets. For example, '[?]' matches the character '?'.

Note that the filename separator ('/' on Unix) is not special to this module. See module glob for pathname expansion (glob uses filter() to match pathname segments). Similarly, filenames starting with a period are not special for this module, and are matched by the * and ? patterns.

fnmatch.fnmatch(filename, pattern)
: Test whether the filename string matches the pattern string, returning True or False. Both parameters are case-normalized using os.path.normcase(). fnmatchcase() can be used to perform a case-sensitive comparison, regardless of whether that's standard for the operating system.

  This example will print all file names in the current directory with the extension .txt:

  ```
  import fnmatch
  import os

  for file in os.listdir('.'):
      if fnmatch.fnmatch(file, '*.txt'):
          print file
  ```

fnmatch.fnmatchcase(filename, pattern)
: Test whether filename matches pattern, returning True or False; the comparison is case-sensitive and does not apply os.path.normcase().

fnmatch.filter(names, pattern)
: Return the subset of the list of names that match pattern. It is the same as [n for n in names if fnmatch(n, pattern)], but implemented more efficiently.

  New in version 2.2.

fnmatch.translate(pattern)
Return the shell-style pattern converted to a regular expression for using with re.match().

Example:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'.*\\.txt\\Z(?ms)'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<_sre.SRE_Match object at 0x...>
```

See also:

Module glob Unix shell-style path expansion.

10.9 linecache — Random access to text lines

Source code: Lib/linecache.py

The linecache module allows one to get any line from any file, while attempting to optimize internally, using a cache, the common case where many lines are read from a single file. This is used by the traceback module to retrieve source lines for inclusion in the formatted traceback.

The linecache module defines the following functions:

linecache.getline(filename, lineno[, module_globals])
Get line lineno from file named filename. This function will never raise an exception — it will return '' on errors (the terminating newline character will be included for lines that are found).

If a file named filename is not found, the function will look for it in the module search path, sys.path, after first checking for a PEP 302 __loader__ in module_globals, in case the module was imported from a zipfile or other non-filesystem import source.

New in version 2.5: The module_globals parameter was added.

linecache.clearcache()
Clear the cache. Use this function if you no longer need lines from files previously read using getline().

linecache.checkcache([filename])
Check the cache for validity. Use this function if files in the cache may have changed on disk, and you require the updated version. If filename is omitted, it will check all the entries in the cache.

Example:

```
>>> import linecache
>>> linecache.getline('/etc/passwd', 4)
'sys:x:3:3:sys:/dev:/bin/sh\n'
```

10.10 shutil — High-level file operations

Source code: Lib/shutil.py

The shutil module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal. For operations on individual files, see also the os module.

> Warning: Even the higher-level file copying functions (shutil.copy(), shutil.copy2()) can't copy all file metadata.
>
> On POSIX platforms, this means that file owner and group are lost as well as ACLs. On Mac OS, the resource fork and other metadata are not used. This means that resources will be lost and file type and creator codes will not be correct. On Windows, file owners, ACLs and alternate data streams are not copied.

10.10.1 Directory and files operations

shutil.copyfileobj(fsrc, fdst[, length])
: Copy the contents of the file-like object fsrc to the file-like object fdst. The integer length, if given, is the buffer size. In particular, a negative length value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption. Note that if the current file position of the fsrc object is not 0, only the contents from the current file position to the end of the file will be copied.

shutil.copyfile(src, dst)
: Copy the contents (no metadata) of the file named src to a file named dst. dst must be the complete target file name; look at shutil.copy() for a copy that accepts a target directory path. If src and dst are the same files, Error is raised. The destination location must be writable; otherwise, an IOError exception will be raised. If dst already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function. src and dst are path names given as strings.

shutil.copymode(src, dst)
: Copy the permission bits from src to dst. The file contents, owner, and group are unaffected. src and dst are path names given as strings.

shutil.copystat(src, dst)
: Copy the permission bits, last access time, last modification time, and flags from src to dst. The file contents, owner, and group are unaffected. src and dst are path names given as strings.

shutil.copy(src, dst)
: Copy the file src to the file or directory dst. If dst is a directory, a file with the same basename as src is created (or overwritten) in the directory specified. Permission bits are copied. src and dst are path names given as strings.

shutil.copy2(src, dst)
: Identical to copy() except that copy2() also attempts to preserve file metadata.

  copy2() uses copystat() to copy the file metadata. Please see copystat() for more information.

shutil.ignore_patterns(*patterns)
: This factory function creates a function that can be used as a callable for copytree()'s ignore argument, ignoring files and directories that match one of the glob-style patterns provided. See the example below.

  New in version 2.6.

shutil.copytree(src, dst, symlinks=False, ignore=None)

Recursively copy an entire directory tree rooted at src. The destination directory, named by dst, must not already exist; it will be created as well as missing parent directories. Permissions and times of directories are copied with copystat(), individual files are copied using shutil.copy2().

If symlinks is true, symbolic links in the source tree are represented as symbolic links in the new tree, but the metadata of the original links is NOT copied; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

If ignore is given, it must be a callable that will receive as its arguments the directory being visited by copytree(), and a list of its contents, as returned by os.listdir(). Since copytree() is called recursively, the ignore callable will be called once for each directory that is copied. The callable must return a sequence of directory and file names relative to the current directory (i.e. a subset of the items in its second argument); these names will then be ignored in the copy process. ignore_patterns() can be used to create such a callable that ignores names based on glob-style patterns.

If exception(s) occur, an Error is raised with a list of reasons.

The source code for this should be considered an example rather than the ultimate tool.

Changed in version 2.3: Error is raised if any exceptions occur during copying, rather than printing a message.

Changed in version 2.5: Create intermediate directories needed to create dst, rather than raising an error. Copy permissions and times of directories using copystat().

Changed in version 2.6: Added the ignore argument to be able to influence what is being copied.

shutil.rmtree(path[, ignore_errors[, onerror]])

Delete an entire directory tree; path must point to a directory (but not a symbolic link to a directory). If ignore_errors is true, errors resulting from failed removals will be ignored; if false or omitted, such errors are handled by calling a handler specified by onerror or, if that is omitted, they raise an exception.

If onerror is provided, it must be a callable that accepts three parameters: function, path, and excinfo. The first parameter, function, is the function which raised the exception; it will be os.path.islink(), os.listdir(), os.remove() or os.rmdir(). The second parameter, path, will be the path name passed to function. The third parameter, excinfo, will be the exception information return by sys.exc_info(). Exceptions raised by onerror will not be caught.

Changed in version 2.6: Explicitly check for path being a symbolic link and raise OSError in that case.

shutil.move(src, dst)

Recursively move a file or directory (src) to another location (dst).

If the destination is an existing directory, then src is moved inside that directory. If the destination already exists but is not a directory, it may be overwritten depending on os.rename() semantics.

If the destination is on the current filesystem, then os.rename() is used. Otherwise, src is copied (using shutil.copy2()) to dst and then removed.

New in version 2.3.

exception shutil.Error

This exception collects exceptions that are raised during a multi-file operation. For copytree(), the exception argument is a list of 3-tuples (srcname, dstname, exception).

New in version 2.3.

copytree example

This example is the implementation of the copytree() function, described above, with the docstring omitted. It demonstrates many of the other functions provided by this module.

```
def copytree(src, dst, symlinks=False, ignore=None):
    names = os.listdir(src)
    if ignore is not None:
        ignored_names = ignore(src, names)
    else:
        ignored_names = set()

    os.makedirs(dst)
    errors = []
    for name in names:
        if name in ignored_names:
            continue
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks, ignore)
            else:
                copy2(srcname, dstname)
            # XXX What about devices, sockets etc.?
        except (IOError, os.error) as why:
            errors.append((srcname, dstname, str(why)))
        # catch the Error from the recursive copytree so that we can
        # continue with other files
        except Error as err:
            errors.extend(err.args[0])
    try:
        copystat(src, dst)
    except WindowsError:
        # can't copy file access times on Windows
        pass
    except OSError as why:
        errors.extend((src, dst, str(why)))
    if errors:
        raise Error(errors)
```

Another example that uses the ignore_patterns() helper:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

This will copy everything except .pyc files and files or directories whose name starts with tmp.

Another example that uses the ignore argument to add a logging call:

```
from shutil import copytree
import logging

def _logpath(path, names):
```

(continues on next page)

(continued from previous page)

```
    logging.info('Working in %s' % path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

10.10.2 Archiving operations

High-level utilities to create and read compressed and archived files are also provided. They rely on the zipfile and tarfile modules.

shutil.make_archive(base_name, format[, root_dir[, base_dir[, verbose[, dry_run[, owner[, group[, logger]]]]]]])

Create an archive file (eg. zip or tar) and returns its name.

base_name is the name of the file to create, including the path, minus any format-specific extension. format is the archive format: one of "zip" (if the zlib module or external zip executable is available), "tar", "gztar" (if the zlib module is available), or "bztar" (if the bz2 module is available).

root_dir is a directory that will be the root directory of the archive; ie. we typically chdir into root_dir before creating the archive.

base_dir is the directory where we start archiving from; ie. base_dir will be the common prefix of all files and directories in the archive.

root_dir and base_dir both default to the current directory.

owner and group are used when creating a tar archive. By default, uses the current owner and group.

logger must be an object compatible with PEP 282, usually an instance of logging.Logger.

New in version 2.7.

shutil.get_archive_formats()

Return a list of supported formats for archiving. Each element of the returned sequence is a tuple (name, description).

By default shutil provides these formats:

- zip: ZIP file (if the zlib module or external zip executable is available).
- tar: uncompressed tar file.
- gztar: gzip'ed tar-file (if the zlib module is available).
- bztar: bzip2'ed tar-file (if the bz2 module is available).

You can register new formats or provide your own archiver for any existing formats, by using register_archive_format().

New in version 2.7.

shutil.register_archive_format(name, function[, extra_args[, description]])

Register an archiver for the format name. function is a callable that will be used to invoke the archiver.

If given, extra_args is a sequence of (name, value) that will be used as extra keywords arguments when the archiver callable is used.

description is used by get_archive_formats() which returns the list of archivers. Defaults to an empty list.

New in version 2.7.

shutil.unregister_archive_format(name)
Remove the archive format name from the list of supported formats.

New in version 2.7.

Archiving example

In this example, we create a gzip'ed tar-file archive containing all files found in the .ssh directory of the user:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

The resulting archive contains:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10 ./known_hosts
```

10.11 dircache — Cached directory listings

Deprecated since version 2.6: The dircache module has been removed in Python 3.

The dircache module defines a function for reading directory listing using a cache, and cache invalidation using the mtime of the directory. Additionally, it defines a function to annotate directories by appending a slash.

The dircache module defines the following functions:

dircache.reset()
Resets the directory cache.

dircache.listdir(path)
Return a directory listing of path, as gotten from os.listdir(). Note that unless path changes, further call to listdir() will not re-read the directory structure.

Note that the list returned should be regarded as read-only. (Perhaps a future version should change it to return a tuple?)

dircache.opendir(path)
Same as listdir(). Defined for backwards compatibility.

dircache.annotate(head, list)
Assume list is a list of paths relative to head, and append, in place, a '/' to each path which points to a directory.

```
>>> import dircache
>>> a = dircache.listdir('/')
>>> a = a[:] # Copy the return value so we can change 'a'
>>> a
['bin', 'boot', 'cdrom', 'dev', 'etc', 'floppy', 'home', 'initrd', 'lib', 'lost+
found', 'mnt', 'proc', 'root', 'sbin', 'tmp', 'usr', 'var', 'vmlinuz']
>>> dircache.annotate('/', a)
>>> a
['bin/', 'boot/', 'cdrom/', 'dev/', 'etc/', 'floppy/', 'home/', 'initrd/', 'lib/
', 'lost+found/', 'mnt/', 'proc/', 'root/', 'sbin/', 'tmp/', 'usr/', 'var/', 'vm
linuz']
```

10.12 macpath — Mac OS 9 path manipulation functions

This module is the Mac OS 9 (and earlier) implementation of the os.path module. It can be used to manipulate old-style Macintosh pathnames on Mac OS X (or any other platform).

The following functions are available in this module: normcase(), normpath(), isabs(), join(), split(), isdir(), isfile(), walk(), exists(). For other functions available in os.path dummy counterparts are available.

See also:

Section File Objects A description of Python's built-in file objects.

Module os Operating system interfaces, including functions to work with files at a lower level than the built-in file object.

CHAPTER

ELEVEN

DATA PERSISTENCE

The modules described in this chapter support storing Python data in a persistent form on disk. The pickle and marshal modules can turn many Python data types into a stream of bytes and then recreate the objects from the bytes. The various DBM-related modules support a family of hash-based file formats that store a mapping of strings to other strings. The bsddb module also provides such disk-based string-to-string mappings based on hashing, and also supports B-Tree and record-based formats.

The list of modules described in this chapter is:

11.1 pickle — Python object serialization

The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as "serialization", "marshalling,"[1] or "flattening", however, to avoid confusion, the terms used here are "pickling" and "unpickling".

This documentation describes both the pickle module and the cPickle module.

> Warning: The pickle module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

11.1.1 Relationship to other Python modules

The pickle module has an optimized cousin called the cPickle module. As its name implies, cPickle is written in C, so it can be up to 1000 times faster than pickle. However it does not support subclassing of the Pickler() and Unpickler() classes, because in cPickle these are functions, not classes. Most applications have no need for this functionality, and can benefit from the improved performance of cPickle. Other than that, the interfaces of the two modules are nearly identical; the common interface is described in this manual and differences are pointed out where necessary. In the following discussions, we use the term "pickle" to collectively describe the pickle and cPickle modules.

The data streams the two modules produce are guaranteed to be interchangeable.

Python has a more primitive serialization module called marshal, but in general pickle should always be the preferred way to serialize Python objects. marshal exists primarily to support Python's .pyc files.

The pickle module differs from marshal in several significant ways:

[1] Don't confuse this with the marshal module

- The pickle module keeps track of the objects it has already serialized, so that later references to the same object won't be serialized again. marshal doesn't do this.

  This has implications both for recursive objects and object sharing. Recursive objects are objects that contain references to themselves. These are not handled by marshal, and in fact, attempting to marshal recursive objects will crash your Python interpreter. Object sharing happens when there are multiple references to the same object in different places in the object hierarchy being serialized. pickle stores such objects only once, and ensures that all other references point to the master copy. Shared objects remain shared, which can be very important for mutable objects.

- marshal cannot be used to serialize user-defined classes and their instances. pickle can save and restore class instances transparently, however the class definition must be importable and live in the same module as when the object was stored.

- The marshal serialization format is not guaranteed to be portable across Python versions. Because its primary job in life is to support .pyc files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise. The pickle serialization format is guaranteed to be backwards compatible across Python releases.

Note that serialization is a more primitive notion than persistence; although pickle reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) issue of concurrent access to persistent objects. The pickle module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. Perhaps the most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The module shelve provides a simple interface to pickle and unpickle objects on DBM-style database files.

11.1.2 Data stream format

The data format used by pickle is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as XDR (which can't represent pointer sharing); however it means that non-Python programs may not be able to reconstruct pickled Python objects.

By default, the pickle data format uses a printable ASCII representation. This is slightly more voluminous than a binary representation. The big advantage of using printable ASCII (and of some other characteristics of pickle's representation) is that for debugging or recovery purposes it is possible for a human to read the pickled file with a standard text editor.

There are currently 3 different protocols which can be used for pickling.

- Protocol version 0 is the original ASCII protocol and is backwards compatible with earlier versions of Python.

- Protocol version 1 is the old binary format which is also compatible with earlier versions of Python.

- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of new-style classes.

Refer to PEP 307 for more information.

If a protocol is not specified, protocol 0 is used. If protocol is specified as a negative value or HIGHEST_PROTOCOL, the highest protocol version available will be used.

Changed in version 2.3: Introduced the protocol parameter.

A binary format, which is slightly more efficient, can be chosen by specifying a protocol version >= 1.

11.1.3 Usage

To serialize an object hierarchy, you first create a pickler, then you call the pickler's dump() method. To de-serialize a data stream, you first create an unpickler, then you call the unpickler's load() method. The pickle module provides the following constant:

pickle.HIGHEST_PROTOCOL
The highest protocol version available. This value can be passed as a protocol value.

New in version 2.3.

Note: Be sure to always open pickle files created with protocols >= 1 in binary mode. For the old ASCII-based pickle protocol 0 you can use either text mode or binary mode as long as you stay consistent.

A pickle file written with protocol 0 in binary mode will contain lone linefeeds as line terminators and therefore will look "funny" when viewed in Notepad or other editors which do not support this format.

The pickle module provides the following functions to make the pickling process more convenient:

pickle.dump(obj, file[, protocol])
Write a pickled representation of obj to the open file object file. This is equivalent to Pickler(file, protocol).dump(obj).

If the protocol parameter is omitted, protocol 0 is used. If protocol is specified as a negative value or HIGHEST_PROTOCOL, the highest protocol version will be used.

Changed in version 2.3: Introduced the protocol parameter.

file must have a write() method that accepts a single string argument. It can thus be a file object opened for writing, a StringIO object, or any other custom object that meets this interface.

pickle.load(file)
Read a string from the open file object file and interpret it as a pickle data stream, reconstructing and returning the original object hierarchy. This is equivalent to Unpickler(file).load().

file must have two methods, a read() method that takes an integer argument, and a readline() method that requires no arguments. Both methods should return a string. Thus file can be a file object opened for reading, a StringIO object, or any other custom object that meets this interface.

This function automatically determines whether the data stream was written in binary mode or not.

pickle.dumps(obj[, protocol])
Return the pickled representation of the object as a string, instead of writing it to a file.

If the protocol parameter is omitted, protocol 0 is used. If protocol is specified as a negative value or HIGHEST_PROTOCOL, the highest protocol version will be used.

Changed in version 2.3: The protocol parameter was added.

pickle.loads(string)
Read a pickled object hierarchy from a string. Characters in the string past the pickled object's representation are ignored.

The pickle module also defines three exceptions:

exception pickle.PickleError
A common base class for the other exceptions defined below. This inherits from Exception.

exception pickle.PicklingError
This exception is raised when an unpicklable object is passed to the dump() method.

exception pickle.UnpicklingError
: This exception is raised when there is a problem unpickling an object. Note that other exceptions may also be raised during unpickling, including (but not necessarily limited to) AttributeError, EOFError, ImportError, and IndexError.

The pickle module also exports two callables[2], Pickler and Unpickler:

class pickle.Pickler(file[, protocol])
: This takes a file-like object to which it will write a pickle data stream.

  If the protocol parameter is omitted, protocol 0 is used. If protocol is specified as a negative value or HIGHEST_PROTOCOL, the highest protocol version will be used.

  Changed in version 2.3: Introduced the protocol parameter.

  file must have a write() method that accepts a single string argument. It can thus be an open file object, a StringIO object, or any other custom object that meets this interface.

  Pickler objects define one (or two) public methods:

  dump(obj)
  : Write a pickled representation of obj to the open file object given in the constructor. Either the binary or ASCII format will be used, depending on the value of the protocol argument passed to the constructor.

  clear_memo()
  : Clears the pickler's "memo". The memo is the data structure that remembers which objects the pickler has already seen, so that shared or recursive objects pickled by reference and not by value. This method is useful when re-using picklers.

    Note: Prior to Python 2.3, clear_memo() was only available on the picklers created by cPickle. In the pickle module, picklers have an instance variable called memo which is a Python dictionary. So to clear the memo for a pickle module pickler, you could do the following:

    ```
    mypickler.memo.clear()
    ```

    Code that does not need to support older versions of Python should simply use clear_memo().

It is possible to make multiple calls to the dump() method of the same Pickler instance. These must then be matched to the same number of calls to the load() method of the corresponding Unpickler instance. If the same object is pickled by multiple dump() calls, the load() will all yield references to the same object.[3]

Unpickler objects are defined as:

class pickle.Unpickler(file)
: This takes a file-like object from which it will read a pickle data stream. This class automatically determines whether the data stream was written in binary mode or not, so it does not need a flag as in the Pickler factory.

  file must have two methods, a read() method that takes an integer argument, and a readline() method that requires no arguments. Both methods should return a string. Thus file can be a file object opened for reading, a StringIO object, or any other custom object that meets this interface.

[2] In the pickle module these callables are classes, which you could subclass to customize the behavior. However, in the cPickle module these callables are factory functions and so cannot be subclassed. One common reason to subclass is to control what objects can actually be unpickled. See section Subclassing Unpicklers for more details.

[3] Warning: this is intended for pickling multiple objects without intervening modifications to the objects or their parts. If you modify an object and then pickle it again using the same Pickler instance, the object is not pickled again — a reference to it is pickled and the Unpickler will return the old value, not the modified one. There are two problems here: (1) detecting changes, and (2) marshalling a minimal set of changes. Garbage Collection may also become a problem here.

Unpickler objects have one (or two) public methods:

load()
: Read a pickled object representation from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein.

  This method automatically determines whether the data stream was written in binary mode or not.

noload()
: This is just like load() except that it doesn't actually create any objects. This is useful primarily for finding what's called "persistent ids" that may be referenced in a pickle data stream. See section The pickle protocol below for more details.

  Note: the noload() method is currently only available on Unpickler objects created with the cPickle module. pickle module Unpicklers do not have the noload() method.

11.1.4 What can be pickled and unpickled?

The following types can be pickled:

- None, True, and False
- integers, long integers, floating point numbers, complex numbers
- normal and Unicode strings
- tuples, lists, sets, and dictionaries containing only picklable objects
- functions defined at the top level of a module
- built-in functions defined at the top level of a module
- classes that are defined at the top level of a module
- instances of such classes whose __dict__ or the result of calling __getstate__() is picklable (see section The pickle protocol for details).

Attempts to pickle unpicklable objects will raise the PicklingError exception; when this happens, an unspecified number of bytes may have already been written to the underlying file. Trying to pickle a highly recursive data structure may exceed the maximum recursion depth, a RuntimeError will be raised in this case. You can carefully raise this limit with sys.setrecursionlimit().

Note that functions (built-in and user-defined) are pickled by "fully qualified" name reference, not by value. This means that only the function name is pickled, along with the name of the module the function is defined in. Neither the function's code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised.[4]

Similarly, classes are pickled by named reference, so the same restrictions in the unpickling environment apply. Note that none of the class's code or data is pickled, so in the following example the class attribute attr is not restored in the unpickling environment:

```
class Foo:
    attr = 'a class attr'

picklestring = pickle.dumps(Foo)
```

[4] The exception raised will likely be an ImportError or an AttributeError but it could be something else.

These restrictions are why picklable functions and classes must be defined in the top level of a module.

Similarly, when class instances are pickled, their class's code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods to the class and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class's __setstate__() method.

11.1.5 The pickle protocol

This section describes the "pickling protocol" that defines the interface between the pickler/unpickler and the objects that are being serialized. This protocol provides a standard way for you to define, customize, and control how your objects are serialized and de-serialized. The description in this section doesn't cover specific customizations that you can employ to make the unpickling environment slightly safer from untrusted pickle data streams; see section Subclassing Unpicklers for more details.

Pickling and unpickling normal class instances

object.__getinitargs__()
: When a pickled class instance is unpickled, its __init__() method is normally not invoked. If it is desirable that the __init__() method be called on unpickling, an old-style class can define a method __getinitargs__(), which should return a tuple of positional arguments to be passed to the class constructor (__init__() for example). Keyword arguments are not supported. The __getinitargs__() method is called at pickle time; the tuple it returns is incorporated in the pickle for the instance.

object.__getnewargs__()
: New-style types can provide a __getnewargs__() method that is used for protocol 2. Implementing this method is needed if the type establishes some internal invariants when the instance is created, or if the memory allocation is affected by the values passed to the __new__() method for the type (as it is for tuples and strings). Instances of a new-style class C are created using

```
obj = C.__new__(C, *args)
```

: where args is the result of calling __getnewargs__() on the original object; if there is no __getnewargs__(), an empty tuple is assumed.

object.__getstate__()
: Classes can further influence how their instances are pickled; if the class defines the method __getstate__(), it is called and the return state is pickled as the contents for the instance, instead of the contents of the instance's dictionary. If there is no __getstate__() method, the instance's __dict__ is pickled.

object.__setstate__(state)
: Upon unpickling, if the class also defines the method __setstate__(), it is called with the unpickled state.[5] If there is no __setstate__() method, the pickled state must be a dictionary and its items are assigned to the new instance's dictionary. If a class defines both __getstate__() and __setstate__(), the state object needn't be a dictionary and these methods can do what they want.[6]

Note: For new-style classes, if __getstate__() returns a false value, the __setstate__() method will not be called.

---

[5] These methods can also be used to implement copying class instances.
[6] This protocol is also used by the shallow and deep copying operations defined in the copy module.

Note: At unpickling time, some methods like __getattr__(), __getattribute__(), or __setattr__() may be called upon the instance. In case those methods rely on some internal invariant being true, the type should implement either __getinitargs__() or __getnewargs__() to establish such an invariant; otherwise, neither __new__() nor __init__() will be called.

Pickling and unpickling extension types

object.__reduce__()

When the Pickler encounters an object of a type it knows nothing about — such as an extension type — it looks in two places for a hint of how to pickle it. One alternative is for the object to implement a __reduce__() method. If provided, at pickling time __reduce__() will be called with no arguments, and it must return either a string or a tuple.

If a string is returned, it names a global variable whose contents are pickled as normal. The string returned by __reduce__() should be the object's local name relative to its module; the pickle module searches the module namespace to determine the object's module.

When a tuple is returned, it must be between two and five elements long. Optional elements can either be omitted, or None can be provided as their value. The contents of this tuple are pickled as normal and used to reconstruct the object at unpickling time. The semantics of each element are:

- A callable object that will be called to create the initial version of the object. The next element of the tuple will provide arguments for this callable, and later elements provide additional state information that will subsequently be used to fully reconstruct the pickled data.

  In the unpickling environment this object must be either a class, a callable registered as a "safe constructor" (see below), or it must have an attribute __safe_for_unpickling__ with a true value. Otherwise, an UnpicklingError will be raised in the unpickling environment. Note that as usual, the callable itself is pickled by name.

- A tuple of arguments for the callable object.

  Changed in version 2.5: Formerly, this argument could also be None.

- Optionally, the object's state, which will be passed to the object's __setstate__() method as described in section Pickling and unpickling normal class instances. If the object has no __setstate__() method, then, as above, the value must be a dictionary and it will be added to the object's __dict__.

- Optionally, an iterator (and not a sequence) yielding successive list items. These list items will be pickled, and appended to the object using either obj.append(item) or obj.extend(list_of_items). This is primarily used for list subclasses, but may be used by other classes as long as they have append() and extend() methods with the appropriate signature. (Whether append() or extend() is used depends on which pickle protocol version is used as well as the number of items to append, so both must be supported.)

- Optionally, an iterator (not a sequence) yielding successive dictionary items, which should be tuples of the form (key, value). These items will be pickled and stored to the object using obj[key] = value. This is primarily used for dictionary subclasses, but may be used by other classes as long as they implement __setitem__().

object.__reduce_ex__(protocol)

It is sometimes useful to know the protocol version when implementing __reduce__(). This can be done by implementing a method named __reduce_ex__() instead of __reduce__(). __reduce_ex__(), when it exists, is called in preference over __reduce__() (you may still provide __re-

duce__() for backwards compatibility). The __reduce_ex__() method will be called with a single integer argument, the protocol version.

The object class implements both __reduce__() and __reduce_ex__(); however, if a subclass overrides __reduce__() but not __reduce_ex__(), the __reduce_ex__() implementation detects this and calls __reduce__().

An alternative to implementing a __reduce__() method on the object to be pickled, is to register the callable with the copy_reg module. This module provides a way for programs to register "reduction functions" and constructors for user-defined types. Reduction functions have the same semantics and interface as the __reduce__() method described above, except that they are called with a single argument, the object to be pickled.

The registered constructor is deemed a "safe constructor" for purposes of unpickling as described above.

Pickling and unpickling external objects

For the benefit of object persistence, the pickle module supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a "persistent id", which is just an arbitrary string of printable ASCII characters. The resolution of such names is not defined by the pickle module; it will delegate this resolution to user defined functions on the pickler and unpickler.[7]

To define external persistent id resolution, you need to set the persistent_id attribute of the pickler object and the persistent_load attribute of the unpickler object.

To pickle objects that have an external persistent id, the pickler must have a custom persistent_id() method that takes an object as an argument and returns either None or the persistent id for that object. When None is returned, the pickler simply pickles the object as normal. When a persistent id string is returned, the pickler will pickle that string, along with a marker so that the unpickler will recognize the string as a persistent id.

To unpickle external objects, the unpickler must have a custom persistent_load() function that takes a persistent id string and returns the referenced object.

Here's a silly example that might shed more light:

```
import pickle
from cStringIO import StringIO

src = StringIO()
p = pickle.Pickler(src)

def persistent_id(obj):
    if hasattr(obj, 'x'):
        return 'the value %d' % obj.x
    else:
        return None

p.persistent_id = persistent_id

class Integer:
    def __init__(self, x):
        self.x = x
    def __str__(self):
        return 'My name is integer %d' % self.x
```

(continues on next page)

[7] The actual mechanism for associating these user defined functions is slightly different for pickle and cPickle. The description given here works the same for both implementations. Users of the pickle module could also use subclassing to effect the same results, overriding the persistent_id() and persistent_load() methods in the derived classes.

(continued from previous page)

```
i = Integer(7)
print i
p.dump(i)

datastream = src.getvalue()
print repr(datastream)
dst = StringIO(datastream)

up = pickle.Unpickler(dst)

class FancyInteger(Integer):
    def __str__(self):
        return 'I am the integer %d' % self.x

def persistent_load(persid):
    if persid.startswith('the value '):
        value = int(persid.split()[2])
        return FancyInteger(value)
    else:
        raise pickle.UnpicklingError, 'Invalid persistent id'

up.persistent_load = persistent_load

j = up.load()
print j
```

In the cPickle module, the unpickler's persistent_load attribute can also be set to a Python list, in which case, when the unpickler reaches a persistent id, the persistent id string will simply be appended to this list. This functionality exists so that a pickle data stream can be "sniffed" for object references without actually instantiating all the objects in a pickle.[8] Setting persistent_load to a list is usually used in conjunction with the noload() method on the Unpickler.

11.1.6 Subclassing Unpicklers

By default, unpickling will import any class that it finds in the pickle data. You can control exactly what gets unpickled and what gets called by customizing your unpickler. Unfortunately, exactly how you do this is different depending on whether you're using pickle or cPickle.[9]

In the pickle module, you need to derive a subclass from Unpickler, overriding the load_global() method. load_global() should read two lines from the pickle data stream where the first line will the name of the module containing the class and the second line will be the name of the instance's class. It then looks up the class, possibly importing the module and digging out the attribute, then it appends what it finds to the unpickler's stack. Later on, this class will be assigned to the __class__ attribute of an empty class, as a way of magically creating an instance without calling its class's __init__(). Your job (should you choose to accept it), would be to have load_global() push onto the unpickler's stack, a known safe version of any class you deem safe to unpickle. It is up to you to produce such a class. Or you could raise an error if you want to disallow all unpickling of instances. If this sounds like a hack, you're right. Refer to the source code to make this work.

Things are a little cleaner with cPickle, but not by much. To control what gets unpickled, you can set the

[8] We'll leave you with the image of Guido and Jim sitting around sniffing pickles in their living rooms.

[9] A word of caution: the mechanisms described here use internal attributes and methods, which are subject to change in future versions of Python. We intend to someday provide a common interface for controlling this behavior, which will work in either pickle or cPickle.

unpickler's find_global attribute to a function or None. If it is None then any attempts to unpickle instances will raise an UnpicklingError. If it is a function, then it should accept a module name and a class name, and return the corresponding class object. It is responsible for looking up the class and performing any necessary imports, and it may raise an error to prevent instances of the class from being unpickled.

The moral of the story is that you should be really careful about the source of the strings your application unpickles.

11.1.7 Example

For the simplest code, use the dump() and load() functions. Note that a self-referencing list is pickled and restored correctly.

```
import pickle

data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

The following example reads the resulting pickled data. When reading a pickle-containing file, you should open the file in binary mode because you can't be sure if the ASCII or binary format was used.

```
import pprint, pickle

pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)

pkl_file.close()
```

Here's a larger example that shows how to modify pickling behavior for a class. The TextReader class opens a text file, and returns the line number and line contents each time its readline() method is called. If a TextReader instance is pickled, all attributes except the file object member are saved. When the instance is unpickled, the file is reopened, and reading resumes from the last location. The __setstate__() and __getstate__() methods are used to implement this behavior.

```
#!/usr/local/bin/python

class TextReader:
```

(continues on next page)

(continued from previous page)

```
    """Print and number lines in a text file."""
    def __init__(self, file):
        self.file = file
        self.fh = open(file)
        self.lineno = 0

    def readline(self):
        self.lineno = self.lineno + 1
        line = self.fh.readline()
        if not line:
            return None
        if line.endswith("\n"):
            line = line[:-1]
        return "%d: %s" % (self.lineno, line)

    def __getstate__(self):
        odict = self.__dict__.copy() # copy the dict since we change it
        del odict['fh']              # remove filehandle entry
        return odict

    def __setstate__(self, dict):
        fh = open(dict['file'])      # reopen file
        count = dict['lineno']       # read from file...
        while count:                 # until line count is restored
            fh.readline()
            count = count - 1
        self.__dict__.update(dict)   # update attributes
        self.fh = fh                 # save the file object
```

A sample usage might be something like this:

```
>>> import TextReader
>>> obj = TextReader.TextReader("TextReader.py")
>>> obj.readline()
'1: #!/usr/local/bin/python'
>>> obj.readline()
'2: '
>>> obj.readline()
'3: class TextReader:'
>>> import pickle
>>> pickle.dump(obj, open('save.p', 'wb'))
```

If you want to see that pickle works across Python processes, start another Python session, before continuing. What follows can happen from either the same process or a new process.

```
>>> import pickle
>>> reader = pickle.load(open('save.p', 'rb'))
>>> reader.readline()
'4:     """Print and number lines in a text file."""'
```

See also:

Module copy_reg Pickle interface constructor registration for extension types.

Module shelve Indexed databases of objects; uses pickle.

Module copy Shallow and deep object copying.

Module marshal High-performance serialization of built-in types.

11.2 cPickle — A faster pickle

The cPickle module supports serialization and de-serialization of Python objects, providing an interface and functionality nearly identical to the pickle module. There are several differences, the most important being performance and subclassability.

First, cPickle can be up to 1000 times faster than pickle because the former is implemented in C. Second, in the cPickle module the callables Pickler() and Unpickler() are functions, not classes. This means that you cannot use them to derive custom pickling and unpickling subclasses. Most applications have no need for this functionality and should benefit from the greatly improved performance of the cPickle module.

The pickle data stream produced by pickle and cPickle are identical, so it is possible to use pickle and cPickle interchangeably with existing pickles.[10]

There are additional minor differences in API between cPickle and pickle, however for most applications, they are interchangeable. More documentation is provided in the pickle module documentation, which includes a list of the documented differences.

11.3 copy_reg — Register pickle support functions

Note: The copy_reg module has been renamed to copyreg in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

The copy_reg module offers a way to define functions used while pickling specific objects. The pickle, cPickle, and copy modules use those functions when pickling/copying those objects. The module provides configuration information about object constructors which are not classes. Such constructors may be factory functions or class instances.

copy_reg.constructor(object)
: Declares object to be a valid constructor. If object is not callable (and hence not valid as a constructor), raises TypeError.

copy_reg.pickle(type, function[, constructor])
: Declares that function should be used as a "reduction" function for objects of type type; type must not be a "classic" class object. (Classic classes are handled differently; see the documentation for the pickle module for details.) function should return either a string or a tuple containing two or three elements.

  The optional constructor parameter, if provided, is a callable object which can be used to reconstruct the object when called with the tuple of arguments returned by function at pickling time. TypeError will be raised if object is a class or constructor is not callable.

  See the pickle module for more details on the interface expected of function and constructor.

11.3.1 Example

The example below would like to show how to register a pickle function and how it will be used:

[10] Since the pickle data format is actually a tiny stack-oriented programming language, and some freedom is taken in the encodings of certain objects, it is possible that the two modules produce different data streams for the same input objects. However it is guaranteed that they will always be able to read each other's data streams.

```
>>> import copy_reg, copy, pickle
>>> class C(object):
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copy_reg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

11.4 shelve — Python object persistence

Source code: Lib/shelve.py

A "shelf" is a persistent, dictionary-like object. The difference with "dbm" databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects — anything that the pickle module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

shelve.open(filename, flag='c', protocol=None, writeback=False)

Open a persistent dictionary. The filename specified is the base filename for the underlying database. As a side-effect, an extension may be added to the filename and more than one file may be created. By default, the underlying database file is opened for reading and writing. The optional flag parameter has the same interpretation as the flag parameter of anydbm.open().

By default, version 0 pickles are used to serialize values. The version of the pickle protocol can be specified with the protocol parameter.

Changed in version 2.3: The protocol parameter was added.

Because of Python semantics, a shelf cannot know when a mutable persistent-dictionary entry is modified. By default modified objects are written only when assigned to the shelf (see Example). If the optional writeback parameter is set to True, all entries accessed are also cached in memory, and written back on sync() and close(); this can make it handier to mutate mutable entries in the persistent dictionary, but, if many entries are accessed, it can consume vast amounts of memory for the cache, and it can make the close operation very slow since all accessed entries are written back (there is no way to determine which accessed entries are mutable, nor which ones were actually mutated).

Like file objects, shelve objects should be closed explicitly to ensure that the persistent data is flushed to disk.

> Warning: Because the shelve module is backed by pickle, it is insecure to load a shelf from an untrusted source. Like with pickle, loading a shelf can execute arbitrary code.

Shelf objects support most of the methods supported by dictionaries. This eases the transition from dictionary based scripts to those requiring persistent storage.

Note, the Python 3 transition methods (viewkeys(), viewvalues(), and viewitems()) are not supported.

Two additional methods are supported:

Shelf.sync()
Write back all entries in the cache if the shelf was opened with writeback set to True. Also empty the cache and synchronize the persistent dictionary on disk, if feasible. This is called automatically when the shelf is closed with close().

Shelf.close()
Synchronize and close the persistent dict object. Operations on a closed shelf will fail with a ValueError.

See also:

Persistent dictionary recipe with widely supported storage formats and having the speed of native dictionaries.

11.4.1 Restrictions

- The choice of which database package will be used (such as dbm, gdbm or bsddb) depends on which interface is available. Therefore it is not safe to open the database directly using dbm. The database is also (unfortunately) subject to the limitations of dbm, if it is used — this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.
- The shelve module does not support concurrent read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program has a shelf open for writing, no other program should have it open for reading or writing. Unix file locking can be used to solve this, but this differs across Unix versions and requires knowledge about the database implementation used.

class shelve.Shelf(dict, protocol=None, writeback=False)
A subclass of UserDict.DictMixin which stores pickled values in the dict object.

By default, version 0 pickles are used to serialize values. The version of the pickle protocol can be specified with the protocol parameter. See the pickle documentation for a discussion of the pickle protocols.

Changed in version 2.3: The protocol parameter was added.

If the writeback parameter is True, the object will hold a cache of all entries accessed and write them back to the dict at sync and close times. This allows natural operations on mutable entries, but can consume much more memory and make sync and close take a long time.

class shelve.BsdDbShelf(dict, protocol=None, writeback=False)
A subclass of Shelf which exposes first(), next(), previous(), last() and set_location() which are available in the bsddb module but not in other database modules. The dict object passed to the constructor must support those methods. This is generally accomplished by calling one of bsddb.hashopen(), bsddb.btopen() or bsddb.rnopen(). The optional protocol and writeback parameters have the same interpretation as for the Shelf class.

class shelve.DbfilenameShelf(filename, flag='c', protocol=None, writeback=False)
A subclass of Shelf which accepts a filename instead of a dict-like object. The underlying file will be opened using anydbm.open(). By default, the file will be created and opened for both read and write. The optional flag parameter has the same interpretation as for the open() function. The optional protocol and writeback parameters have the same interpretation as for the Shelf class.

11.4.2 Example

To summarize the interface (key is a string, data is an arbitrary object):

```
import shelve

d = shelve.open(filename) # open -- file may get suffix added by low-level
                          # library

d[key] = data   # store data at key (overwrites old data if
                # using an existing key)
data = d[key]   # retrieve a COPY of data at key (raise KeyError if no
                # such key)
del d[key]      # delete data stored at key (raises KeyError
                # if no such key)
flag = d.has_key(key)   # true if the key exists
klist = d.keys() # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = range(4)  # this works as expected, but...
d['xx'].append(5)   # *this doesn't!* -- d['xx'] is STILL range(4)!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']      # extracts the copy
temp.append(5)      # mutates the copy
d['xx'] = temp      # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()       # close it
```

See also:

Module anydbm Generic interface to dbm-style databases.

Module bsddb BSD db database interface.

Module dbhash Thin layer around the bsddb which provides an open() function like the other database modules.

Module dbm Standard Unix database interface.

Module dumbdbm Portable implementation of the dbm interface.

Module gdbm GNU database interface, based on the dbm interface.

Module pickle Object serialization used by shelve.

Module cPickle High-performance version of pickle.

11.5 marshal — Internal Python object serialization

This module contains functions that can read and write Python values in a binary format. The format is specific to Python, but independent of machine architecture issues (e.g., you can write a Python value to a

file on a PC, transport the file to a Sun, and read it back there). Details of the format are undocumented on purpose; it may change between Python versions (although it rarely does).[1]

This is not a general "persistence" module. For general persistence and transfer of Python objects through RPC calls, see the modules pickle and shelve. The marshal module exists mainly to support reading and writing the "pseudo-compiled" code for Python modules of .pyc files. Therefore, the Python maintainers reserve the right to modify the marshal format in backward incompatible ways should the need arise. If you're serializing and de-serializing Python objects, use the pickle module instead – the performance is comparable, version independence is guaranteed, and pickle supports a substantially wider range of objects than marshal.

> Warning: The marshal module is not intended to be secure against erroneous or maliciously constructed data. Never unmarshal data received from an untrusted or unauthenticated source.

Not all Python object types are supported; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported: booleans, integers, long integers, floating point numbers, complex numbers, strings, Unicode objects, tuples, lists, sets, frozensets, dictionaries, and code objects, where it should be understood that tuples, lists, sets, frozensets and dictionaries are only supported as long as the values contained therein are themselves supported; and recursive lists, sets and dictionaries should not be written (they will cause infinite loops). The singletons None, Ellipsis and StopIteration can also be marshalled and unmarshalled.

> Warning: On machines where C's long int type has more than 32 bits (such as the DEC Alpha), it is possible to create plain Python integers that are longer than 32 bits. If such an integer is marshaled and read back in on a machine where C's long int type has only 32 bits, a Python long integer object is returned instead. While of a different type, the numeric value is the same. (This behavior is new in Python 2.2. In earlier versions, all but the least-significant 32 bits of the value were lost, and a warning message was printed.)

There are functions that read/write files as well as functions operating on strings.

The module defines these functions:

marshal.dump(value, file[, version])
: Write the value on the open file. The value must be a supported type. The file must be an open file object such as sys.stdout or returned by open() or os.popen(). It may not be a wrapper such as TemporaryFile on Windows. It must be opened in binary mode (`'wb'` or `'w+b'`).

  If the value has (or contains an object that has) an unsupported type, a ValueError exception is raised — but garbage data will also be written to the file. The object will not be properly read back by load().

  New in version 2.4: The version argument indicates the data format that dump should use (see below).

marshal.load(file)
: Read one value from the open file and return it. If no valid value is read (e.g. because the data has a different Python version's incompatible marshal format), raise EOFError, ValueError or TypeError. The file must be an open file object opened in binary mode (`'rb'` or `'r+b'`).

  Note: If an object containing an unsupported type was marshalled with dump(), load() will substitute None for the unmarshallable type.

  ---

[1] The name of this module stems from a bit of terminology used by the designers of Modula-3 (amongst others), who use the term "marshalling" for shipping of data around in a self-contained form. Strictly speaking, "to marshal" means to convert some data from internal to external form (in an RPC buffer for instance) and "unmarshalling" for the reverse process.

marshal.dumps(value[, version])
Return the string that would be written to a file by dump(value, file). The value must be a supported type. Raise a ValueError exception if value has (or contains an object that has) an unsupported type.

New in version 2.4: The version argument indicates the data format that dumps should use (see below).

marshal.loads(string)
Convert the string to a value. If no valid value is found, raise EOFError, ValueError or TypeError. Extra characters in the string are ignored.

In addition, the following constants are defined:

marshal.version
Indicates the format that the module uses. Version 0 is the historical format, version 1 (added in Python 2.4) shares interned strings and version 2 (added in Python 2.5) uses a binary format for floating point numbers. The current version is 2.

New in version 2.4.

11.6 anydbm — Generic access to DBM-style databases

Note: The anydbm module has been renamed to dbm in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

anydbm is a generic interface to variants of the DBM database — dbhash (requires bsddb), gdbm, or dbm. If none of these modules is installed, the slow-but-simple implementation in module dumbdbm will be used.

anydbm.open(filename[, flag[, mode]])
Open the database file filename and return a corresponding object.

If the database file already exists, the whichdb module is used to determine its type and the appropriate module is used; if it does not exist, the first module listed above that can be imported is used.

The optional flag argument must be one of these values:

Value	Meaning
`'r'`	Open existing database for reading only (default)
`'w'`	Open existing database for reading and writing
`'c'`	Open database for reading and writing, creating it if it doesn't exist
`'n'`	Always create a new, empty database, open for reading and writing

If not specified, the default value is `'r'`.

The optional mode argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0666 (and will be modified by the prevailing umask).

exception anydbm.error
A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception also named anydbm.error as the first item — the latter is used when anydbm.error is raised.

The object returned by open() supports most of the same functionality as dictionaries; keys and their corresponding values can be stored, retrieved, and deleted, and the has_key() and keys() methods are available. Keys and values must always be strings.

The following example records some hostnames and a corresponding title, and then prints out the contents of the database:

```
import anydbm

# Open database, creating it if necessary.
db = anydbm.open('cache', 'c')

# Record some values
db['www.python.org'] = 'Python Website'
db['www.cnn.com'] = 'Cable News Network'

# Loop through contents.  Other dictionary methods
# such as .keys(), .values() also work.
for k, v in db.iteritems():
    print k, '\t', v

# Storing a non-string key or value will raise an exception (most
# likely a TypeError).
db['www.yahoo.com'] = 4

# Close when done.
db.close()
```

In addition to the dictionary-like methods, anydbm objects provide the following method:

anydbm.close()
: Close the anydbm database.

See also:

Module dbhash BSD db database interface.

Module dbm Standard Unix database interface.

Module dumbdbm Portable implementation of the dbm interface.

Module gdbm GNU database interface, based on the dbm interface.

Module shelve General object persistence built on top of the Python dbm interface.

Module whichdb Utility module used to determine the type of an existing database.

11.7 whichdb — Guess which DBM module created a database

Note: The whichdb module's only function has been put into the dbm module in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

The single function in this module attempts to guess which of the several simple database modules available—dbm, gdbm, or dbhash—should be used to open a given file.

whichdb.whichdb(filename)
: Returns one of the following values: None if the file can't be opened because it's unreadable or doesn't exist; the empty string (`''`) if the file's format can't be guessed; or a string containing the required module name, such as `'dbm'` or `'gdbm'`.

11.8 dbm — Simple "database" interface

Note: The dbm module has been renamed to dbm.ndbm in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

The dbm module provides an interface to the Unix "(n)dbm" library. Dbm objects behave like mappings (dictionaries), except that keys and values are always strings. Printing a dbm object doesn't print the keys and values, and the items() and values() methods are not supported.

This module can be used with the "classic" ndbm interface, the BSD DB compatibility interface, or the GNU GDBM compatibility interface. On Unix, the configure script will attempt to locate the appropriate header file to simplify building this module.

The module defines the following:

exception dbm.error
: Raised on dbm-specific errors, such as I/O errors. KeyError is raised for general mapping errors like specifying an incorrect key.

dbm.library
: Name of the ndbm implementation library used.

dbm.open(filename[, flag[, mode]])
: Open a dbm database and return a dbm object. The filename argument is the name of the database file (without the .dir or .pag extensions; note that the BSD DB implementation of the interface will append the extension .db and only create one file).

  The optional flag argument must be one of these values:

Value	Meaning
`'r'`	Open existing database for reading only (default)
`'w'`	Open existing database for reading and writing
`'c'`	Open database for reading and writing, creating it if it doesn't exist
`'n'`	Always create a new, empty database, open for reading and writing

  The optional mode argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0666 (and will be modified by the prevailing umask).

  In addition to the dictionary-like methods, dbm objects provide the following method:

  dbm.close()
  : Close the dbm database.

See also:

Module anydbm Generic interface to dbm-style databases.

Module gdbm Similar interface to the GNU GDBM library.

Module whichdb Utility module used to determine the type of an existing database.

11.9 gdbm — GNU's reinterpretation of dbm

Note: The gdbm module has been renamed to dbm.gnu in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

This module is quite similar to the dbm module, but uses gdbm instead to provide some additional functionality. Please note that the file formats created by gdbm and dbm are incompatible.

The gdbm module provides an interface to the GNU DBM library. gdbm objects behave like mappings (dictionaries), except that keys and values are always strings. Printing a gdbm object doesn't print the keys and values, and the items() and values() methods are not supported.

The module defines the following constant and functions:

exception gdbm.error
: Raised on gdbm-specific errors, such as I/O errors. KeyError is raised for general mapping errors like specifying an incorrect key.

gdbm.open(filename[, flag[, mode]])
: Open a gdbm database and return a gdbm object. The filename argument is the name of the database file.

  The optional flag argument can be:

Value	Meaning
`'r'`	Open existing database for reading only (default)
`'w'`	Open existing database for reading and writing
`'c'`	Open database for reading and writing, creating it if it doesn't exist
`'n'`	Always create a new, empty database, open for reading and writing

  The following additional characters may be appended to the flag to control how the database is opened:

Value	Meaning
`'f'`	Open the database in fast mode. Writes to the database will not be synchronized.
`'s'`	Synchronized mode. This will cause changes to the database to be immediately written to the file.
`'u'`	Do not lock database.

  Not all flags are valid for all versions of gdbm. The module constant open_flags is a string of supported flag characters. The exception error is raised if an invalid flag is specified.

  The optional mode argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0666.

In addition to the dictionary-like methods, gdbm objects have the following methods:

gdbm.firstkey()
: It's possible to loop over every key in the database using this method and the nextkey() method. The traversal is ordered by gdbm's internal hash values, and won't be sorted by the key values. This method returns the starting key.

gdbm.nextkey(key)
: Returns the key that follows key in the traversal. The following code prints every key in the database db, without having to create a list in memory that contains them all:

```
k = db.firstkey()
while k != None:
```

(continues on next page)

(continued from previous page)

```
    print k
    k = db.nextkey(k)
```

gdbm.reorganize()
: If you have carried out a lot of deletions and would like to shrink the space used by the gdbm file, this routine will reorganize the database. gdbm will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

gdbm.sync()
: When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

gdbm.close()
: Close the gdbm database.

See also:

Module anydbm Generic interface to dbm-style databases.

Module whichdb Utility module used to determine the type of an existing database.

11.10 dbhash — DBM-style interface to the BSD database library

Deprecated since version 2.6: The dbhash module has been removed in Python 3.

The dbhash module provides a function to open databases using the BSD db library. This module mirrors the interface of the other Python database modules that provide access to DBM-style databases. The bsddb module is required to use dbhash.

This module provides an exception and a function:

exception dbhash.error
: Exception raised on database errors other than KeyError. It is a synonym for bsddb.error.

dbhash.open(path[, flag[, mode]])
: Open a db database and return the database object. The path argument is the name of the database file.

  The flag argument can be:

Value	Meaning
`'r'`	Open existing database for reading only (default)
`'w'`	Open existing database for reading and writing
`'c'`	Open database for reading and writing, creating it if it doesn't exist
`'n'`	Always create a new, empty database, open for reading and writing

  For platforms on which the BSD db library supports locking, an `'l'` can be appended to indicate that locking should be used.

  The optional mode parameter is used to indicate the Unix permission bits that should be set if a new database must be created; this will be masked by the current umask value for the process.

See also:

Module anydbm Generic interface to dbm-style databases.

Module bsddb Lower-level interface to the BSD db library.

Module whichdb Utility module used to determine the type of an existing database.

11.10.1 Database Objects

The database objects returned by open() provide the methods common to all the DBM-style databases and mapping objects. The following methods are available in addition to the standard methods.

dbhash.first()
: It's possible to loop over every key/value pair in the database using this method and the next() method. The traversal is ordered by the databases internal hash values, and won't be sorted by the key values. This method returns the starting key.

dbhash.last()
: Return the last key/value pair in a database traversal. This may be used to begin a reverse-order traversal; see previous().

dbhash.next()
: Returns the key next key/value pair in a database traversal. The following code prints every key in the database db, without having to create a list in memory that contains them all:

```
print db.first()
for i in xrange(1, len(db)):
    print db.next()
```

dbhash.previous()
: Returns the previous key/value pair in a forward-traversal of the database. In conjunction with last(), this may be used to implement a reverse-order traversal.

dbhash.sync()
: This method forces any unwritten data to be written to the disk.

11.11 bsddb — Interface to Berkeley DB library

Deprecated since version 2.6: The bsddb module has been removed in Python 3.

The bsddb module provides an interface to the Berkeley DB library. Users can create hash, btree or record based library files using the appropriate open call. Bsddb objects behave generally like dictionaries. Keys and values must be strings, however, so to use other objects as keys or to store other kinds of objects the user must serialize them somehow, typically using marshal.dumps() or pickle.dumps().

The bsddb module requires a Berkeley DB library version from 4.0 thru 4.7.

See also:

http://www.jcea.es/programacion/pybsddb.htm The website with documentation for the bsddb.db Python Berkeley DB interface that closely mirrors the object oriented interface provided in Berkeley DB 4.x itself.

http://www.oracle.com/database/berkeley-db/ The Berkeley DB library.

A more modern DB, DBEnv and DBSequence object interface is available in the bsddb.db module which closely matches the Berkeley DB C API documented at the above URLs. Additional features provided by the bsddb.db API include fine tuning, transactions, logging, and multiprocess concurrent database access.

The following is a description of the legacy bsddb interface compatible with the old Python bsddb module. Starting in Python 2.5 this interface should be safe for multithreaded access. The bsddb.db API is recommended for threading users as it provides better control.

The bsddb module defines the following functions that create objects that access the appropriate type of Berkeley DB file. The first two arguments of each function are the same. For ease of portability, only the first two arguments should be used in most instances.

bsddb.hashopen(filename[, flag[, mode[, pgsize[, ffactor[, nelem[, cachesize[, lorder[, hflags]]]]]]]])
Open the hash format file named filename. Files never intended to be preserved on disk may be created by passing None as the filename. The optional flag identifies the mode used to open the file. It may be 'r' (read only), 'w' (read-write), 'c' (read-write - create if necessary; the default) or 'n' (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level dbopen() function. Consult the Berkeley DB documentation for their use and interpretation.

bsddb.btopen(filename[, flag[, mode[, btflags[, cachesize[, maxkeypage[, minkeypage[, pgsize[, lorder]]]]]]]])
Open the btree format file named filename. Files never intended to be preserved on disk may be created by passing None as the filename. The optional flag identifies the mode used to open the file. It may be 'r' (read only), 'w' (read-write), 'c' (read-write - create if necessary; the default) or 'n' (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level dbopen function. Consult the Berkeley DB documentation for their use and interpretation.

bsddb.rnopen(filename[, flag[, mode[, rnflags[, cachesize[, pgsize[, lorder[, rlen[, delim[, source[, pad]]]]]]]]]])
Open a DB record format file named filename. Files never intended to be preserved on disk may be created by passing None as the filename. The optional flag identifies the mode used to open the file. It may be 'r' (read only), 'w' (read-write), 'c' (read-write - create if necessary; the default) or 'n' (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level dbopen function. Consult the Berkeley DB documentation for their use and interpretation.

Note: Beginning in 2.3 some Unix versions of Python may have a bsddb185 module. This is present only to allow backwards compatibility with systems which ship with the old Berkeley DB 1.85 database library. The bsddb185 module should never be used directly in new code. The module has been removed in Python 3. If you find you still need it look in PyPI.

See also:

Module dbhash DBM-style interface to the bsddb

11.11.1 Hash, BTree and Record Objects

Once instantiated, hash, btree and record objects support the same methods as dictionaries. In addition, they support the methods listed below.

Changed in version 2.3.1: Added dictionary methods.

bsddbobject.close()
Close the underlying file. The object can no longer be accessed. Since there is no open open() method for these objects, to open the file again a new bsddb module open function must be called.

bsddbobject.keys()
Return the list of keys contained in the DB file. The order of the list is unspecified and should not be relied on. In particular, the order of the list returned is different for different file formats.

bsddbobject.has_key(key)
: Return 1 if the DB file contains the argument as a key.

bsddbobject.set_location(key)
: Set the cursor to the item indicated by key and return a tuple containing the key and its value. For binary tree databases (opened using btopen()), if key does not actually exist in the database, the cursor will point to the next item in sorted order and return that key and value. For other databases, KeyError will be raised if key is not found in the database.

bsddbobject.first()
: Set the cursor to the first item in the DB file and return it. The order of keys in the file is unspecified, except in the case of B-Tree databases. This method raises bsddb.error if the database is empty.

bsddbobject.next()
: Set the cursor to the next item in the DB file and return it. The order of keys in the file is unspecified, except in the case of B-Tree databases.

bsddbobject.previous()
: Set the cursor to the previous item in the DB file and return it. The order of keys in the file is unspecified, except in the case of B-Tree databases. This is not supported on hashtable databases (those opened with hashopen()).

bsddbobject.last()
: Set the cursor to the last item in the DB file and return it. The order of keys in the file is unspecified. This is not supported on hashtable databases (those opened with hashopen()). This method raises bsddb.error if the database is empty.

bsddbobject.sync()
: Synchronize the database on disk.

Example:

```
>>> import bsddb
>>> db = bsddb.btopen('spam.db', 'c')
>>> for i in range(10): db['%d'%i] = '%d'% (i*i)
...
>>> db['3']
'9'
>>> db.keys()
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> db.first()
('0', '0')
>>> db.next()
('1', '1')
>>> db.last()
('9', '81')
>>> db.set_location('2')
('2', '4')
>>> db.previous()
('1', '1')
>>> for k, v in db.iteritems():
...     print k, v
0 0
1 1
2 4
3 9
4 16
5 25
6 36
```

(continues on next page)

(continued from previous page)

```
7 49
8 64
9 81
>>> '8' in db
True
>>> db.sync()
0
```

11.12 dumbdbm — Portable DBM implementation

Note: The dumbdbm module has been renamed to dbm.dumb in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

Note: The dumbdbm module is intended as a last resort fallback for the anydbm module when no more robust module is available. The dumbdbm module is not written for speed and is not nearly as heavily used as the other database modules.

The dumbdbm module provides a persistent dictionary-like interface which is written entirely in Python. Unlike other modules such as gdbm and bsddb, no external library is required. As with other persistent mappings, the keys and values must always be strings.

The module defines the following:

exception dumbdbm.error
: Raised on dumbdbm-specific errors, such as I/O errors. KeyError is raised for general mapping errors like specifying an incorrect key.

dumbdbm.open(filename[, flag[, mode]])
: Open a dumbdbm database and return a dumbdbm object. The filename argument is the basename of the database file (without any specific extensions). When a dumbdbm database is created, files with .dat and .dir extensions are created.

  The optional flag argument is currently ignored; the database is always opened for update, and will be created if it does not exist.

  The optional mode argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0666 (and will be modified by the prevailing umask).

  Changed in version 2.2: The mode argument was ignored in earlier versions.

In addition to the dictionary-like methods, dumbdm objects provide the following method:

dumbdbm.close()
: Close the dumbdm database.

See also:

Module anydbm Generic interface to dbm-style databases.

Module dbm Similar interface to the DBM/NDBM library.

Module gdbm Similar interface to the GNU GDBM library.

Module shelve Persistence module which stores non-string data.

Module whichdb Utility module used to determine the type of an existing database.

11.12.1 Dumbdbm Objects

In addition to the methods provided by the UserDict.DictMixin class, dumbdbm objects provide the following methods.

dumbdbm.sync()
: Synchronize the on-disk directory and data files. This method is called by the sync() method of Shelve objects.

11.13 sqlite3 — DB-API 2.0 interface for SQLite databases

New in version 2.5.

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The sqlite3 module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249.

To use the module, you must first create a Connection object that represents the database. Here the data will be stored in the example.db file:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

You can also supply the special name :memory: to create a database in RAM.

Once you have a Connection, you can create a Cursor object and call its execute() method to perform SQL commands:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
             (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

The data you've saved is persistent and is available in subsequent sessions:

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
```

Usually your SQL operations will need to use values from Python variables. You shouldn't assemble your query using Python's string operations because doing so is insecure; it makes your program vulnerable to an SQL injection attack (see https://xkcd.com/327/ for humorous example of what can go wrong).

Instead, use the DB-API's parameter substitution. Put ? as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor's execute() method. (Other database modules may use a different placeholder, such as %s or :1.) For example:

```
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)

# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print c.fetchone()

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
             ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
             ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
            ]
c.executemany('INSERT INTO stocks VALUES (?,?,?,?,?)', purchases)
```

To retrieve data after executing a SELECT statement, you can either treat the cursor as an iterator, call the cursor's fetchone() method to retrieve a single matching row, or call fetchall() to get a list of the matching rows.

This example uses the iterator form:

```
>>> for row in c.execute('SELECT * FROM stocks ORDER BY price'):
        print row

(u'2006-01-05', u'BUY', u'RHAT', 100, 35.14)
(u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
(u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
(u'2006-04-05', u'BUY', u'MSFT', 1000, 72.0)
```

See also:

https://github.com/ghaering/pysqlite The pysqlite web page – sqlite3 is developed externally under the name "pysqlite".

https://www.sqlite.org The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

http://www.w3schools.com/sql/ Tutorial, reference and examples for learning SQL syntax.

PEP 249 - Database API Specification 2.0 PEP written by Marc-André Lemburg.

11.13.1 Module functions and constants

sqlite3.version
: The version number of this module, as a string. This is not the version of the SQLite library.

sqlite3.version_info
: The version number of this module, as a tuple of integers. This is not the version of the SQLite library.

sqlite3.sqlite_version
The version number of the run-time SQLite library, as a string.

sqlite3.sqlite_version_info
The version number of the run-time SQLite library, as a tuple of integers.

sqlite3.PARSE_DECLTYPES
This constant is meant to be used with the detect_types parameter of the connect() function.

Setting it makes the sqlite3 module parse the declared type for each column it returns. It will parse out the first word of the declared type, i. e. for "integer primary key", it will parse out "integer", or for "number(10)" it will parse out "number". Then for that column, it will look into the converters dictionary and use the converter function registered for that type there.

sqlite3.PARSE_COLNAMES
This constant is meant to be used with the detect_types parameter of the connect() function.

Setting this makes the SQLite interface parse the column name for each column it returns. It will look for a string formed [mytype] in there, and then decide that 'mytype' is the type of the column. It will try to find an entry of 'mytype' in the converters dictionary and then use the converter function found there to return the value. The column name found in Cursor.description is only the first word of the column name, i. e. if you use something like 'as "x [datetime]"' in your SQL, then we will parse out everything until the first blank for the column name: the column name would simply be "x".

sqlite3.connect(database[, timeout, detect_types, isolation_level, check_same_thread, factory, cached_statements])
Opens a connection to the SQLite database file database. You can use ":memory:" to open a database connection to a database that resides in RAM instead of on disk.

When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The timeout parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds).

For the isolation_level parameter, please see the Connection.isolation_level property of Connection objects.

SQLite natively supports only the types TEXT, INTEGER, REAL, BLOB and NULL. If you want to use other types you must add support for them yourself. The detect_types parameter and the using custom converters registered with the module-level register_converter() function allow you to easily do that.

detect_types defaults to 0 (i. e. off, no type detection), you can set it to any combination of PARSE_DECLTYPES and PARSE_COLNAMES to turn type detection on.

By default, the sqlite3 module uses its Connection class for the connect call. You can, however, subclass the Connection class and make connect() use your class instead by providing your class for the factory parameter.

Consult the section SQLite and Python types of this manual for details.

The sqlite3 module internally uses a statement cache to avoid SQL parsing overhead. If you want to explicitly set the number of statements that are cached for the connection, you can set the cached_statements parameter. The currently implemented default is to cache 100 statements.

sqlite3.register_converter(typename, callable)
Registers a callable to convert a bytestring from the database into a custom Python type. The callable will be invoked for all database values that are of the type typename. Confer the parameter detect_types of the connect() function for how the type detection works. Note that typename and the name of the type in your query are matched in case-insensitive manner.

sqlite3.register_adapter(type, callable)
Registers a callable to convert the custom Python type type into one of SQLite's supported types. The callable callable accepts as single parameter the Python value, and must return a value of the following types: int, long, float, str (UTF-8 encoded), unicode or buffer.

sqlite3.complete_statement(sql)
Returns True if the string sql contains one or more complete SQL statements terminated by semicolons. It does not verify that the SQL is syntactically correct, only that there are no unclosed string literals and the statement is terminated by a semicolon.

This can be used to build a shell for SQLite, as in the following example:

```
# A minimal SQLite shell for experiments

import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print "Enter your SQL commands to execute in sqlite3."
print "Enter a blank line to exit."

while True:
    line = raw_input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print cur.fetchall()
        except sqlite3.Error as e:
            print "An error occurred:", e.args[0]
        buffer = ""

con.close()
```

sqlite3.enable_callback_tracebacks(flag)
By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with flag set to True. Afterwards, you will get tracebacks from callbacks on sys.stderr. Use False to disable the feature again.

11.13.2 Connection Objects

class sqlite3.Connection
A SQLite database connection has the following attributes and methods:

isolation_level
Get or set the current isolation level. None for autocommit mode or one of "DEFERRED", "IMMEDIATE" or "EXCLUSIVE". See section Controlling Transactions for a more detailed explanation.

cursor(factory=Cursor)
: The cursor method accepts a single optional parameter factory. If supplied, this must be a callable returning an instance of Cursor or its subclasses.

commit()
: This method commits the current transaction. If you don't call this method, anything you did since the last call to commit() is not visible from other database connections. If you wonder why you don't see the data you've written to the database, please check you didn't forget to call this method.

rollback()
: This method rolls back any changes to the database since the last call to commit().

close()
: This closes the database connection. Note that this does not automatically call commit(). If you just close your database connection without calling commit() first, your changes will be lost!

execute(sql[, parameters])
: This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's execute method with the parameters given.

executemany(sql[, parameters])
: This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's executemany method with the parameters given.

executescript(sql_script)
: This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's executescript method with the parameters given.

create_function(name, num_params, func)
: Creates a user-defined function that you can later use from within SQL statements under the function name name. num_params is the number of parameters the function accepts, and func is a Python callable that is called as the SQL function.

  The function can return any of the types supported by SQLite: unicode, str, int, long, float, buffer and None.

  Example:

```
import sqlite3
import md5

def md5sum(t):
    return md5.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", ("foo",))
print cur.fetchone()[0]
```

create_aggregate(name, num_params, aggregate_class)
: Creates a user-defined aggregate function.

  The aggregate class must implement a step method, which accepts the number of parameters num_params, and a finalize method which will return the final result of the aggregate.

  The finalize method can return any of the types supported by SQLite: unicode, str, int, long, float, buffer and None.

  Example:

```
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print cur.fetchone()[0]
```

create_collation(name, callable)

Creates a collation with the specified name and callable. The callable will be passed two string arguments. It should return -1 if the first is ordered lower than the second, 0 if they are ordered equal and 1 if the first is ordered higher than the second. Note that this controls sorting (ORDER BY in SQL) so your comparisons don't affect other SQL operations.

Note that the callable will get its parameters as Python bytestrings, which will normally be encoded in UTF-8.

The following example shows a custom collation that sorts "the wrong way":

```
import sqlite3

def collate_reverse(string1, string2):
    return -cmp(string1, string2)

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a",), ("b",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print row
con.close()
```

To remove a collation, call create_collation with None as callable:

```
con.create_collation("reverse", None)
```

interrupt()

You can call this method from a different thread to abort any queries that might be executing on the connection. The query will then abort and the caller will get an exception.

set_authorizer(authorizer_callback)

This routine registers a callback. The callback is invoked for each attempt to access a col-

umn of a table in the database. The callback should return SQLITE_OK if access is allowed, SQLITE_DENY if the entire SQL statement should be aborted with an error and SQLITE_IGNORE if the column should be treated as a NULL value. These constants are available in the sqlite3 module.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or None depending on the first argument. The 4th argument is the name of the database ("main", "temp", etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or None if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the sqlite3 module.

set_progress_handler(handler, n)
This routine registers a callback. The callback is invoked for every n instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with None for handler.

New in version 2.6.

enable_load_extension(enabled)
This routine allows/disallows the SQLite engine to load SQLite extensions from shared libraries. SQLite extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

Loadable extensions are disabled by default. See[1].

New in version 2.7.

```
import sqlite3

con = sqlite3.connect(":memory:")

# enable extension loading
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript("""
    insert into recipe (name, ingredients) values ('broccoli stew', 'broccoli peppers cheese tomatoes');
    insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpkin onions garlic celery');
    insert into recipe (name, ingredients) values ('broccoli pie', 'broccoli cheese onions flour');
```

(continues on next page)

[1] The sqlite3 module is not built with loadable extension support by default, because some platforms (notably Mac OS X) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must modify setup.py and remove the line that sets SQLITE_OMIT_LOAD_EXTENSION.

(continued from previous page)

```
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin sugar flour butter');
    """)
for row in con.execute("select rowid, name, ingredients from recipe where name match 'pie'"):
    print row
```

load_extension(path)
: This routine loads a SQLite extension from a shared library. You have to enable extension loading with enable_load_extension() before you can use this routine.

  Loadable extensions are disabled by default. See[1].

  New in version 2.7.

row_factory
: You can change this attribute to a callable that accepts the cursor and the original row as a tuple and will return the real result row. This way, you can implement more advanced ways of returning results, such as returning an object that can also access columns by name.

  Example:

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print cur.fetchone()["a"]
```

  If returning a tuple doesn't suffice and you want name-based access to columns, you should consider setting row_factory to the highly-optimized sqlite3.Row type. Row provides both index-based and case-insensitive name-based access to columns with almost no memory overhead. It will probably be better than your own custom dictionary-based approach or even a db_row based solution.

text_factory
: Using this attribute you can control what objects are returned for the TEXT data type. By default, this attribute is set to unicode and the sqlite3 module will return Unicode objects for TEXT. If you want to return bytestrings instead, you can set it to str.

  For efficiency reasons, there's also a way to return Unicode objects only for non-ASCII data, and bytestrings otherwise. To activate it, set this attribute to sqlite3.OptimizedUnicode.

  You can also set it to any other callable that accepts a single bytestring parameter and returns the resulting object.

  See the following example code for illustration:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
```

(continues on next page)

(continued from previous page)

```
AUSTRIA = u"\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = str
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is str
# the bytestrings will be encoded in UTF-8, unless you stored garbage in the
# database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that will ignore Unicode characters that cannot be
# decoded from UTF-8
con.text_factory = lambda x: unicode(x, "utf-8", "ignore")
cur.execute("select ?", ("this is latin1 and would normally create errors" +
                         u"\xe4\xf6\xfc".encode("latin1"),))
row = cur.fetchone()
assert type(row[0]) is unicode

# sqlite3 offers a built-in optimized text_factory that will return bytestring
# objects, if the data is in ASCII only, and otherwise return unicode objects
con.text_factory = sqlite3.OptimizedUnicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is unicode

cur.execute("select ?", ("Germany",))
row = cur.fetchone()
assert type(row[0]) is str
```

total_changes
: Returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

iterdump
: Returns an iterator to dump the database in an SQL text format. Useful when saving an in-memory database for later restoration. This function provides the same capabilities as the .dump command in the sqlite3 shell.

  New in version 2.6.

  Example:

```
# Convert file existing_db.db to SQL dump file dump.sql
import sqlite3, os

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
```

11.13.3 Cursor Objects

class sqlite3.Cursor

A Cursor instance has the following attributes and methods.

execute(sql[, parameters])

Executes an SQL statement. The SQL statement may be parameterized (i. e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks (qmark style) and named placeholders (named style).

Here's an example of both styles:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table people (name_last, age)")

who = "Yeltsin"
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)", (who, age))

# And this is the named style:
cur.execute("select * from people where name_last=:who and age=:age", {"who": who, "age": age})

print cur.fetchone()
```

execute() will only execute a single SQL statement. If you try to execute more than one statement with it, it will raise a Warning. Use executescript() if you want to execute multiple SQL statements with one call.

executemany(sql, seq_of_parameters)

Executes an SQL command against all parameter sequences or mappings found in the sequence sql. The sqlite3 module also allows using an iterator yielding parameters instead of a sequence.

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def next(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)
```

(continues on next page)

(continued from previous page)

```
cur.execute("select c from characters")
print cur.fetchall()
```

Here's a shorter example using a generator:

```
import sqlite3
import string

def char_generator():
    for c in string.lowercase:
        yield (c,)

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

cur.executemany("insert into characters(c) values (?)", char_generator())

cur.execute("select c from characters")
print cur.fetchall()
```

executescript(sql_script)

This is a nonstandard convenience method for executing multiple SQL statements at once. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter.

sql_script can be a bytestring or a Unicode string.

Example:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
    """)
```

fetchone()

Fetches the next row of a query result set, returning a single sequence, or None when no more data is available.

fetchmany([size=cursor.arraysize])
: Fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available.

  The number of rows to fetch per call is specified by the size parameter. If it is not given, the cursor's arraysize determines the number of rows to be fetched. The method should try to fetch as many rows as indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned.

  Note there are performance considerations involved with the size parameter. For optimal performance, it is usually best to use the arraysize attribute. If the size parameter is used, then it is best for it to retain the same value from one fetchmany() call to the next.

fetchall()
: Fetches all (remaining) rows of a query result, returning a list. Note that the cursor's arraysize attribute can affect the performance of this operation. An empty list is returned when no rows are available.

rowcount
: Although the Cursor class of the sqlite3 module implements this attribute, the database engine's own support for the determination of "rows affected"/"rows selected" is quirky.

  For executemany() statements, the number of modifications are summed up into rowcount.

  As required by the Python DB API Spec, the rowcount attribute "is -1 in case no executeXX() has been performed on the cursor or the rowcount of the last operation is not determinable by the interface". This includes SELECT statements because we cannot determine the number of rows a query produced until all rows were fetched.

  With SQLite versions before 3.6.5, rowcount is set to 0 if you make a DELETE FROM table without any condition.

lastrowid
: This read-only attribute provides the rowid of the last modified row. It is only set if you issued an INSERT statement using the execute() method. For operations other than INSERT or when executemany() is called, lastrowid is set to None.

description
: This read-only attribute provides the column names of the last query. To remain compatible with the Python DB API, it returns a 7-tuple for each column where the last six items of each tuple are None.

  It is set for SELECT statements without any matching rows as well.

connection
: This read-only attribute provides the SQLite database Connection used by the Cursor object. A Cursor object created by calling con.cursor() will have a connection attribute that refers to con:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```

11.13.4 Row Objects

class sqlite3.Row
: A Row instance serves as a highly optimized row_factory for Connection objects. It tries to mimic a tuple in most of its features.

It supports mapping access by column name and index, iteration, representation, equality testing and len().

If two Row objects have exactly the same columns and their members are equal, they compare equal.

Changed in version 2.6: Added iteration and equality (hashability).

keys()

This method returns a list of column names. Immediately after a query, it is the first member of each tuple in Cursor.description.

New in version 2.6.

Let's assume we initialize a table as in the example given above:

```
conn = sqlite3.connect(":memory:")
c = conn.cursor()
c.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')
c.execute("""insert into stocks
          values ('2006-01-05','BUY','RHAT',100,35.14)""")
conn.commit()
c.close()
```

Now we plug Row in:

```
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
<type 'sqlite3.Row'>
>>> r
(u'2006-01-05', u'BUY', u'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
u'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print member
...
2006-01-05
BUY
RHAT
100.0
35.14
```

11.13.5 SQLite and Python types

Introduction

SQLite natively supports the following types: NULL, INTEGER, REAL, TEXT, BLOB.

The following Python types can thus be sent to SQLite without any problem:

Python type	SQLite type
None	NULL
int	INTEGER
long	INTEGER
float	REAL
str (UTF8-encoded)	TEXT
unicode	TEXT
buffer	BLOB

This is how SQLite types are converted to Python types by default:

SQLite type	Python type
NULL	None
INTEGER	int or long, depending on size
REAL	float
TEXT	depends on text_factory, unicode by default
BLOB	buffer

The type system of the sqlite3 module is extensible in two ways: you can store additional Python types in a SQLite database via object adaptation, and you can let the sqlite3 module convert SQLite types to different Python types via converters.

Using adapters to store additional Python types in SQLite databases

As described before, SQLite supports only a limited set of types natively. To use other Python types with SQLite, you must adapt them to one of the sqlite3 module's supported types for SQLite: one of NoneType, int, long, float, str, unicode, buffer.

There are two ways to enable the sqlite3 module to adapt a custom Python type to one of the supported ones.

Letting your object adapt itself

This is a good approach if you write the class yourself. Let's suppose you have a class like this:

```
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y
```

Now you want to store the point in a single SQLite column. First you'll have to choose one of the supported types first to be used for representing the point. Let's just use str and separate the coordinates using a semicolon. Then you need to give your class a method __conform__(self, protocol) which must return the converted value. The parameter protocol will be PrepareProtocol.

```
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

```

(continues on next page)

(continued from previous page)

```
    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print cur.fetchone()[0]
```

Registering an adapter callable

The other possibility is to create a function that converts the type to the string representation and register the function with register_adapter().

Note: The type/class to adapt must be a new-style class, i. e. it must have object as one of its bases.

```
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print cur.fetchone()[0]
```

The sqlite3 module has two default adapters for Python's built-in datetime.date and datetime.datetime types. Now let's suppose we want to store datetime.datetime objects not in ISO representation, but as a Unix timestamp.

```
import sqlite3
import datetime, time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
cur = con.cursor()

now = datetime.datetime.now()
```

(continues on next page)

(continued from previous page)

```
cur.execute("select ?", (now,))
print cur.fetchone()[0]
```

Converting SQLite values to custom Python types

Writing an adapter lets you send custom Python types to SQLite. But to make it really useful we need to make the Python to SQLite to Python roundtrip work.

Enter converters.

Let's go back to the Point class. We stored the x and y coordinates separated via semicolons as strings in SQLite.

First, we'll define a converter function that accepts the string as a parameter and constructs a Point object from it.

Note: Converter functions always get called with a string, no matter under which data type you sent the value to SQLite.

```
def convert_point(s):
    x, y = map(float, s.split(";"))
    return Point(x, y)
```

Now you need to make the sqlite3 module know that what you select from the database is actually a point. There are two ways of doing this:

- Implicitly via the declared type
- Explicitly via the column name

Both ways are described in section Module functions and constants, in the entries for the constants PARSE_DECLTYPES and PARSE_COLNAMES.

The following example illustrates both approaches.

```
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

def convert_point(s):
    x, y = map(float, s.split(";"))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
```

(continues on next page)

(continued from previous page)

```
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

########################
# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print "with declared types:", cur.fetchone()[0]
cur.close()
con.close()

########################
# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print "with column names:", cur.fetchone()[0]
cur.close()
con.close()
```

Default adapters and converters

There are default adapters for the date and datetime types in the datetime module. They will be sent as ISO dates/ISO timestamps to SQLite.

The default converters are registered under the name "date" for datetime.date and under the name "timestamp" for datetime.datetime.

This way, you can use date/timestamps from Python without any additional fiddling in most cases. The format of the adapters is also compatible with the experimental SQLite date/time functions.

The following example demonstrates this.

```
import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print today, "=>", row[0], type(row[0])
print now, "=>", row[1], type(row[1])
```

(continues on next page)

(continued from previous page)

```
cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"')
row = cur.fetchone()
print "current_date", row[0], type(row[0])
print "current_timestamp", row[1], type(row[1])
```

If a timestamp stored in SQLite has a fractional part longer than 6 numbers, its value will be truncated to microsecond precision by the timestamp converter.

11.13.6 Controlling Transactions

By default, the sqlite3 module opens transactions implicitly before a Data Modification Language (DML) statement (i.e. INSERT/UPDATE/DELETE/REPLACE), and commits transactions implicitly before a non-DML, non-query statement (i. e. anything other than SELECT or the aforementioned).

So if you are within a transaction and issue a command like CREATE TABLE ..., VACUUM, PRAGMA, the sqlite3 module will commit implicitly before executing that command. There are two reasons for doing that. The first is that some of these commands don't work within transactions. The other reason is that sqlite3 needs to keep track of the transaction state (if a transaction is active or not).

You can control which kind of BEGIN statements sqlite3 implicitly executes (or none at all) via the isolation_level parameter to the connect() call, or via the isolation_level property of connections.

If you want autocommit mode, then set isolation_level to None.

Otherwise leave it at its default, which will result in a plain "BEGIN" statement, or set it to one of SQLite's supported isolation levels: "DEFERRED", "IMMEDIATE" or "EXCLUSIVE".

11.13.7 Using sqlite3 efficiently

Using shortcut methods

Using the nonstandard execute(), executemany() and executescript() methods of the Connection object, your code can be written more concisely because you don't have to create the (often superfluous) Cursor objects explicitly. Instead, the Cursor objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a SELECT statement and iterate over it directly using only a single call on the Connection object.

```
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
    ]

con = sqlite3.connect(":memory:")

# Create the table
con.execute("create table person(firstname, lastname)")

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)

# Print the table contents
```

(continues on next page)

(continued from previous page)

```
for row in con.execute("select firstname, lastname from person"):
    print row

print "I just deleted", con.execute("delete from person").rowcount, "rows"
```

Accessing columns by name instead of by index

One useful feature of the sqlite3 module is the built-in sqlite3.Row class designed to be used as a row factory.

Rows wrapped with this class can be accessed both by index (like tuples) and case-insensitively by name:

```
import sqlite3

con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
for row in cur:
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]
```

Using the connection as a context manager

New in version 2.6.

Connection objects can be used as context managers that automatically commit or rollback transactions. In the event of an exception, the transaction is rolled back; otherwise, the transaction is committed:

```
import sqlite3

con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key, firstname varchar unique)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))

# con.rollback() is called after the with block finishes with an exception, the
# exception is still raised and must be caught
try:
    with con:
        con.execute("insert into person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
    print "couldn't add Joe twice"
```

11.13.8 Common issues

Multithreading

Older SQLite versions had issues with sharing connections between threads. That's why the Python module disallows sharing connections and cursors between threads. If you still try to do so, you will get an exception at runtime.

The only exception is calling the interrupt() method, which only makes sense to call from a different thread.

CHAPTER

TWELVE

DATA COMPRESSION AND ARCHIVING

The modules described in this chapter support data compression with the zlib, gzip, and bzip2 algorithms, and the creation of ZIP- and tar-format archives. See also Archiving operations provided by the shutil module.

12.1 zlib — Compression compatible with gzip

For applications that require data compression, the functions in this module allow compression and decompression, using the zlib library. The zlib library has its own home page at http://www.zlib.net. There are known incompatibilities between the Python module and versions of the zlib library earlier than 1.1.3; 1.1.3 has a security vulnerability, so we recommend using 1.1.4 or later.

zlib's functions have many options and often need to be used in a particular order. This documentation doesn't attempt to cover all of the permutations; consult the zlib manual at http://www.zlib.net/manual.html for authoritative information.

For reading and writing .gz files see the gzip module.

The available exception and functions in this module are:

exception zlib.error
: Exception raised on compression and decompression errors.

zlib.adler32(data[, value])
: Computes an Adler-32 checksum of data. (An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much more quickly.) If value is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

  This function always returns an integer object.

Note: To generate the same numeric value across all Python versions and platforms use adler32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

Changed in version 2.6: The return value is in the range [-2**31, 2**31-1] regardless of platform. In older versions the value is signed on some platforms and unsigned on others.

Changed in version 3.0: The return value is unsigned and in the range [0, 2**32-1] regardless of platform.

zlib.compress(string[, level])
: Compresses the data in string, returning a string contained compressed data. level is an integer from 0 to 9 controlling the level of compression; 1 is fastest and produces the least compression, 9 is slowest and produces the most. 0 is no compression. The default value is 6. Raises the error exception if any error occurs.

zlib.compressobj([level[, method[, wbits[, memlevel[, strategy]]]]])
: Returns a compression object, to be used for compressing data streams that won't fit into memory at once. level is an integer from 0 to 9 or -1, controlling the level of compression; 1 is fastest and produces the least compression, 9 is slowest and produces the most. 0 is no compression. The default value is -1 (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6).

  method is the compression algorithm. Currently, the only supported value is DEFLATED.

  The wbits argument controls the size of the history buffer (or the "window size") used when compressing data, and whether a header and trailer is included in the output. It can take several ranges of values. The default is 15.

  - +9 to +15: The base-two logarithm of the window size, which therefore ranges between 512 and 32768. Larger values produce better compression at the expense of greater memory usage. The resulting output will include a zlib-specific header and trailer.
  - −9 to −15: Uses the absolute value of wbits as the window size logarithm, while producing a raw output stream with no header or trailing checksum.
  - +25 to +31 = 16 + (9 to 15): Uses the low 4 bits of the value as the window size logarithm, while including a basic gzip header and trailing checksum in the output.

  memlevel controls the amount of memory used for internal compression state. Valid values range from 1 to 9. Higher values using more memory, but are faster and produce smaller output. The default is 8.

  strategy is used to tune the compression algorithm. Possible values are Z_DEFAULT_STRATEGY, Z_FILTERED, and Z_HUFFMAN_ONLY. The default is Z_DEFAULT_STRATEGY.

zlib.crc32(data[, value])
: Computes a CRC (Cyclic Redundancy Check) checksum of data. If value is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

  This function always returns an integer object.

Note: To generate the same numeric value across all Python versions and platforms use crc32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

Changed in version 2.6: The return value is in the range [-2**31, 2**31-1] regardless of platform. In older versions the value would be signed on some platforms and unsigned on others.

Changed in version 3.0: The return value is unsigned and in the range [0, 2**32-1] regardless of platform.

zlib.decompress(string[, wbits[, bufsize]])
: Decompresses the data in string, returning a string containing the uncompressed data. The wbits parameter depends on the format of string, and is discussed further below. If bufsize is given, it is used as the initial size of the output buffer. Raises the error exception if any error occurs.

The wbits parameter controls the size of the history buffer (or "window size"), and what header and trailer format is expected. It is similar to the parameter for compressobj(), but accepts more ranges of values:

- +8 to +15: The base-two logarithm of the window size. The input must include a zlib header and trailer.
- 0: Automatically determine the window size from the zlib header. Only supported since zlib 1.2.3.5.
- −8 to −15: Uses the absolute value of wbits as the window size logarithm. The input must be a raw stream with no header or trailer.
- +24 to +31 = 16 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm. The input must include a gzip header and trailer.
- +40 to +47 = 32 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm, and automatically accepts either the zlib or gzip format.

When decompressing a stream, the window size must not be smaller than the size originally used to compress the stream; using a too-small value may result in an error exception. The default wbits value is 15, which corresponds to the largest window size and requires a zlib header and trailer to be included.

bufsize is the initial size of the buffer used to hold decompressed data. If more space is required, the buffer size will be increased as needed, so you don't have to get this value exactly right; tuning it will only save a few calls to malloc(). The default size is 16384.

zlib.decompressobj([wbits])
Returns a decompression object, to be used for decompressing data streams that won't fit into memory at once.

The wbits parameter controls the size of the history buffer (or the "window size"), and what header and trailer format is expected. It has the same meaning as described for decompress().

Compression objects support the following methods:

Compress.compress(string)
Compress string, returning a string containing compressed data for at least part of the data in string. This data should be concatenated to the output produced by any preceding calls to the compress() method. Some input may be kept in internal buffers for later processing.

Compress.flush([mode])
All pending input is processed, and a string containing the remaining compressed output is returned. mode can be selected from the constants Z_SYNC_FLUSH, Z_FULL_FLUSH, or Z_FINISH, defaulting to Z_FINISH. Z_SYNC_FLUSH and Z_FULL_FLUSH allow compressing further strings of data, while Z_FINISH finishes the compressed stream and prevents compressing any more data. After calling flush() with mode set to Z_FINISH, the compress() method cannot be called again; the only realistic action is to delete the object.

Compress.copy()
Returns a copy of the compression object. This can be used to efficiently compress a set of data that share a common initial prefix.

New in version 2.5.

Decompression objects support the following methods, and two attributes:

Decompress.unused_data
A string which contains any bytes past the end of the compressed data. That is, this remains "" until the last byte that contains compression data is available. If the whole string turned out to contain compressed data, this is "", the empty string.

The only way to determine where a string of compressed data ends is by actually decompressing it. This means that when compressed data is contained part of a larger file, you can only find the end of it by reading data and feeding it followed by some non-empty string into a decompression object's decompress() method until the unused_data attribute is no longer the empty string.

Decompress.unconsumed_tail
A string that contains any data that was not consumed by the last decompress() call because it exceeded the limit for the uncompressed data buffer. This data has not yet been seen by the zlib machinery, so you must feed it (possibly with further data concatenated to it) back to a subsequent decompress() method call in order to get correct output.

Decompress.decompress(string[, max_length])
Decompress string, returning a string containing the uncompressed data corresponding to at least part of the data in string. This data should be concatenated to the output produced by any preceding calls to the decompress() method. Some of the input data may be preserved in internal buffers for later processing.

If the optional parameter max_length is non-zero then the return value will be no longer than max_length. This may mean that not all of the compressed input can be processed; and unconsumed data will be stored in the attribute unconsumed_tail. This string must be passed to a subsequent call to decompress() if decompression is to continue. If max_length is not supplied then the whole input is decompressed, and unconsumed_tail is an empty string.

Decompress.flush([length])
All pending input is processed, and a string containing the remaining uncompressed output is returned. After calling flush(), the decompress() method cannot be called again; the only realistic action is to delete the object.

The optional parameter length sets the initial size of the output buffer.

Decompress.copy()
Returns a copy of the decompression object. This can be used to save the state of the decompressor midway through the data stream in order to speed up random seeks into the stream at a future point.

New in version 2.5.

See also:

Module gzip Reading and writing gzip-format files.

http://www.zlib.net The zlib library home page.

http://www.zlib.net/manual.html The zlib manual explains the semantics and usage of the library's many functions.

12.2 gzip — Support for gzip files

Source code: Lib/gzip.py

This module provides a simple interface to compress and decompress files just like the GNU programs gzip and gunzip would.

The data compression is provided by the zlib module.

The gzip module provides the GzipFile class which is modeled after Python's File Object. The GzipFile class reads and writes gzip-format files, automatically compressing or decompressing the data so that it looks like an ordinary file object.

Note that additional file formats which can be decompressed by the gzip and gunzip programs, such as those produced by compress and pack, are not supported by this module.

The module defines the following items:

class gzip.GzipFile([filename[, mode[, compresslevel[, fileobj[, mtime]]]]])
Constructor for the GzipFile class, which simulates most of the methods of a file object, with the exception of the readinto() and truncate() methods. At least one of fileobj and filename must be given a non-trivial value.

The new class instance is based on fileobj, which can be a regular file, a StringIO object, or any other object which simulates a file. It defaults to None, in which case filename is opened to provide a file object.

When fileobj is not None, the filename argument is only used to be included in the gzip file header, which may include the original filename of the uncompressed file. It defaults to the filename of fileobj, if discernible; otherwise, it defaults to the empty string, and in this case the original filename is not included in the header.

The mode argument can be any of `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, or `'wb'`, depending on whether the file will be read or written. The default is the mode of fileobj if discernible; otherwise, the default is `'rb'`. If not given, the 'b' flag will be added to the mode to ensure the file is opened in binary mode for cross-platform portability.

The compresslevel argument is an integer from 0 to 9 controlling the level of compression; 1 is fastest and produces the least compression, and 9 is slowest and produces the most compression. 0 is no compression. The default is 9.

The mtime argument is an optional numeric timestamp to be written to the stream when compressing. All gzip compressed streams are required to contain a timestamp. If omitted or None, the current time is used. This module ignores the timestamp when decompressing; however, some programs, such as gunzip, make use of it. The format of the timestamp is the same as that of the return value of time.time() and of the st_mtime attribute of the object returned by os.stat().

Calling a GzipFile object's close() method does not close fileobj, since you might wish to append more material after the compressed data. This also allows you to pass a StringIO object opened for writing as fileobj, and retrieve the resulting memory buffer using the StringIO object's getvalue() method.

GzipFile supports iteration and the with statement.

Changed in version 2.7: Support for the with statement was added.

Changed in version 2.7: Support for zero-padded files was added.

New in version 2.7: The mtime argument.

gzip.open(filename[, mode[, compresslevel]])
This is a shorthand for GzipFile(filename, mode, compresslevel). The filename argument is required; mode defaults to `'rb'` and compresslevel defaults to 9.

12.2.1 Examples of usage

Example of how to read a compressed file:

```
import gzip
with gzip.open('file.txt.gz', 'rb') as f:
    file_content = f.read()
```

Example of how to create a compressed GZIP file:

```
import gzip
content = "Lots of content here"
with gzip.open('file.txt.gz', 'wb') as f:
    f.write(content)
```

Example of how to GZIP compress an existing file:

```
import gzip
import shutil
with open('file.txt', 'rb') as f_in, gzip.open('file.txt.gz', 'wb') as f_out:
    shutil.copyfileobj(f_in, f_out)
```

See also:

Module zlib The basic data compression module needed to support the gzip file format.

12.3 bz2 — Compression compatible with bzip2

New in version 2.3.

This module provides a comprehensive interface for the bz2 compression library. It implements a complete file interface, one-shot (de)compression functions, and types for sequential (de)compression.

Here is a summary of the features offered by the bz2 module:

- BZ2File class implements a complete file interface, including readline(), readlines(), writelines(), seek(), etc;
- BZ2File class implements emulated seek() support;
- BZ2File class implements universal newline support;
- BZ2File class offers an optimized line iteration using the readahead algorithm borrowed from file objects;
- Sequential (de)compression supported by BZ2Compressor and BZ2Decompressor classes;
- One-shot (de)compression supported by compress() and decompress() functions;
- Thread safety uses individual locking mechanism.

Note: Handling of multi-stream bzip2 files is not supported. Modules such as bz2file let you overcome this.

12.3.1 (De)compression of files

Handling of compressed files is offered by the BZ2File class.

class bz2.BZ2File(filename[, mode[, buffering[, compresslevel]]])

Open a bz2 file. Mode can be either `'r'` or `'w'`, for reading (default) or writing. When opened for writing, the file will be created if it doesn't exist, and truncated otherwise. If buffering is given, 0 means unbuffered, and larger numbers specify the buffer size; the default is 0. If compresslevel is given, it must be a number between 1 and 9; the default is 9. Add a `'U'` to mode to open the file for input in universal newlines mode. Any line ending in the input file will be seen as a `'\n'` in Python. Also, a file so opened gains the attribute newlines; the value for this attribute is one of None (no newline read yet), `'\r'`, `'\n'`, `'\r\n'` or a tuple containing all the newline types seen. Universal newlines are available only when reading. Instances support iteration in the same way as normal file instances.

BZ2File supports the with statement.

Changed in version 2.7: Support for the with statement was added.

Note: This class does not support input files containing multiple streams (such as those produced by the pbzip2 tool). When reading such an input file, only the first stream will be accessible. If you require support for multi-stream files, consider using the third-party bz2file module (available from PyPI). This module provides a backport of Python 3.3's BZ2File class, which does support multi-stream files.

---

close()
: Close the file. Sets data attribute closed to true. A closed file cannot be used for further I/O operations. close() may be called more than once without error.

read([size])
: Read at most size uncompressed bytes, returned as a string. If the size argument is negative or omitted, read until EOF is reached.

readline([size])
: Return the next line from the file, as a string, retaining newline. A non-negative size argument limits the maximum number of bytes to return (an incomplete line may be returned then). Return an empty string at EOF.

readlines([size])
: Return a list of lines read. The optional size argument, if given, is an approximate bound on the total number of bytes in the lines returned.

xreadlines()
: For backward compatibility. BZ2File objects now include the performance optimizations previously implemented in the xreadlines module.

  Deprecated since version 2.3: This exists only for compatibility with the method by this name on file objects, which is deprecated. Use for line in file instead.

seek(offset[, whence])
: Move to new file position. Argument offset is a byte count. Optional argument whence defaults to os.SEEK_SET or 0 (offset from start of file; offset should be >= 0); other values are os.SEEK_CUR or 1 (move relative to current position; offset can be positive or negative), and os.SEEK_END or 2 (move relative to end of file; offset is usually negative, although many platforms allow seeking beyond the end of a file).

  Note that seeking of bz2 files is emulated, and depending on the parameters the operation may be extremely slow.

tell()
: Return the current file position, an integer (may be a long integer).

write(data)
: Write string data to file. Note that due to buffering, close() may be needed before the file on disk reflects the data written.

writelines(sequence_of_strings)
: Write the sequence of strings to the file. Note that newlines are not added. The sequence can be any iterable object producing strings. This is equivalent to calling write() for each string.

12.3.2 Sequential (de)compression

Sequential compression and decompression is done using the classes BZ2Compressor and BZ2Decompressor.

class bz2.BZ2Compressor([compresslevel])
: Create a new compressor object. This object may be used to compress data sequentially. If you want to compress data in one shot, use the compress() function instead. The compresslevel parameter, if given, must be a number between 1 and 9; the default is 9.

  compress(data)
  : Provide more data to the compressor object. It will return chunks of compressed data whenever possible. When you've finished providing data to compress, call the flush() method to finish the compression process, and return what is left in internal buffers.

  flush()
  : Finish the compression process and return what is left in internal buffers. You must not use the compressor object after calling this method.

class bz2.BZ2Decompressor
: Create a new decompressor object. This object may be used to decompress data sequentially. If you want to decompress data in one shot, use the decompress() function instead.

  decompress(data)
  : Provide more data to the decompressor object. It will return chunks of decompressed data whenever possible. If you try to decompress data after the end of stream is found, EOFError will be raised. If any data was found after the end of stream, it'll be ignored and saved in unused_data attribute.

12.3.3 One-shot (de)compression

One-shot compression and decompression is provided through the compress() and decompress() functions.

bz2.compress(data[, compresslevel])
: Compress data in one shot. If you want to compress data sequentially, use an instance of BZ2Compressor instead. The compresslevel parameter, if given, must be a number between 1 and 9; the default is 9.

bz2.decompress(data)
: Decompress data in one shot. If you want to decompress data sequentially, use an instance of BZ2Decompressor instead.

12.4 zipfile — Work with ZIP archives

New in version 1.6.

Source code: Lib/zipfile.py

The ZIP file format is a common archive and compression standard. This module provides tools to create, read, write, append, and list a ZIP file. Any advanced use of this module will require an understanding of the format, as defined in PKZIP Application Note.

This module does not currently handle multi-disk ZIP files. It can handle ZIP files that use the ZIP64 extensions (that is ZIP files that are more than 4 GByte in size). It supports decryption of encrypted files

in ZIP archives, but it currently cannot create an encrypted file. Decryption is extremely slow as it is implemented in native Python rather than C.

The module defines the following items:

exception zipfile.BadZipfile
The error raised for bad ZIP files (old name: zipfile.error).

exception zipfile.LargeZipFile
The error raised when a ZIP file would require ZIP64 functionality but that has not been enabled.

class zipfile.ZipFile
The class for reading and writing ZIP files. See section ZipFile Objects for constructor details.

class zipfile.PyZipFile
Class for creating ZIP archives containing Python libraries.

class zipfile.ZipInfo([filename[, date_time]])
Class used to represent information about a member of an archive. Instances of this class are returned by the getinfo() and infolist() methods of ZipFile objects. Most users of the zipfile module will not need to create these, but only use those created by this module. filename should be the full name of the archive member, and date_time should be a tuple containing six fields which describe the time of the last modification to the file; the fields are described in section ZipInfo Objects.

zipfile.is_zipfile(filename)
Returns True if filename is a valid ZIP file based on its magic number, otherwise returns False. filename may be a file or file-like object too.

Changed in version 2.7: Support for file and file-like objects.

zipfile.ZIP_STORED
The numeric constant for an uncompressed archive member.

zipfile.ZIP_DEFLATED
The numeric constant for the usual ZIP compression method. This requires the zlib module. No other compression methods are currently supported.

See also:

PKZIP Application Note Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

Info-ZIP Home Page Information about the Info-ZIP project's ZIP archive programs and development libraries.

12.4.1 ZipFile Objects

class zipfile.ZipFile(file[, mode[, compression[, allowZip64]]])
Open a ZIP file, where file can be either a path to a file (a string) or a file-like object. The mode parameter should be `'r'` to read an existing file, `'w'` to truncate and write a new file, or `'a'` to append to an existing file. If mode is `'a'` and file refers to an existing ZIP file, then additional files are added to it. If file does not refer to a ZIP file, then a new ZIP archive is appended to the file. This is meant for adding a ZIP archive to another file (such as python.exe).

Changed in version 2.6: If mode is a and the file does not exist at all, it is created.

compression is the ZIP compression method to use when writing the archive, and should be ZIP_STORED or ZIP_DEFLATED; unrecognized values will cause RuntimeError to be raised. If ZIP_DEFLATED is specified but the zlib module is not available, RuntimeError is also raised. The

default is ZIP_STORED. If allowZip64 is True zipfile will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 2 GB. If it is false (the default) zipfile will raise an exception when the ZIP file would require ZIP64 extensions. ZIP64 extensions are disabled by default because the default zip and unzip commands on Unix (the InfoZIP utilities) don't support these extensions.

Changed in version 2.7.1: If the file is created with mode `'a'` or `'w'` and then closed without adding any files to the archive, the appropriate ZIP structures for an empty archive will be written to the file.

ZipFile is also a context manager and therefore supports the with statement. In the example, myzip is closed after the with statement's suite is finished—even if an exception occurs:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

New in version 2.7: Added the ability to use ZipFile as a context manager.

ZipFile.close()
Close the archive file. You must call close() before exiting your program or essential records will not be written.

ZipFile.getinfo(name)
Return a ZipInfo object with information about the archive member name. Calling getinfo() for a name not currently contained in the archive will raise a KeyError.

ZipFile.infolist()
Return a list containing a ZipInfo object for each member of the archive. The objects are in the same order as their entries in the actual ZIP file on disk if an existing archive was opened.

ZipFile.namelist()
Return a list of archive members by name.

ZipFile.open(name[, mode[, pwd]])
Extract a member from the archive as a file-like object (ZipExtFile). name is the name of the file in the archive, or a ZipInfo object. The mode parameter, if included, must be one of the following: `'r'` (the default), `'U'`, or `'rU'`. Choosing `'U'` or `'rU'` will enable universal newline support in the read-only object. pwd is the password used for encrypted files. Calling open() on a closed ZipFile will raise a RuntimeError.

Note: The file-like object is read-only and provides the following methods: read(), readline(), readlines(), __iter__(), next().

Note: If the ZipFile was created by passing in a file-like object as the first argument to the constructor, then the object returned by open() shares the ZipFile's file pointer. Under these circumstances, the object returned by open() should not be used after any additional operations are performed on the ZipFile object. If the ZipFile was created by passing in a string (the filename) as the first argument to the constructor, then open() will create a new file object that will be held by the ZipExtFile, allowing it to operate independently of the ZipFile.

Note: The open(), read() and extract() methods can take a filename or a ZipInfo object. You will appreciate this when trying to read a ZIP file that contains members with duplicate names.

New in version 2.6.

ZipFile.extract(member[, path[, pwd]])
Extract a member from the archive to the current working directory; member must be its full name or a ZipInfo object). Its file information is extracted as accurately as possible. path specifies a different directory to extract to. member can be a filename or a ZipInfo object. pwd is the password used for encrypted files.

Returns the normalized path created (a directory or new file).

New in version 2.6.

Note: If a member filename is an absolute path, a drive/UNC sharepoint and leading (back)slashes will be stripped, e.g.: ///foo/bar becomes foo/bar on Unix, and C:\foo\bar becomes foo\bar on Windows. And all ".." components in a member filename will be removed, e.g.: ../../foo../../ba..r becomes foo../ba..r. On Windows illegal characters (:, <, >, |, ", ?, and *) replaced by underscore (_).

ZipFile.extractall([path[, members[, pwd]]])
Extract all members from the archive to the current working directory. path specifies a different directory to extract to. members is optional and must be a subset of the list returned by namelist(). pwd is the password used for encrypted files.

Warning: Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of path, e.g. members that have absolute filenames starting with "/" or filenames with two dots "..".

Changed in version 2.7.4: The zipfile module attempts to prevent that. See extract() note.

New in version 2.6.

ZipFile.printdir()
Print a table of contents for the archive to sys.stdout.

ZipFile.setpassword(pwd)
Set pwd as default password to extract encrypted files.

New in version 2.6.

ZipFile.read(name[, pwd])
Return the bytes of the file name in the archive. name is the name of the file in the archive, or a ZipInfo object. The archive must be open for read or append. pwd is the password used for encrypted files and, if specified, it will override the default password set with setpassword(). Calling read() on a closed ZipFile will raise a RuntimeError.

Changed in version 2.6: pwd was added, and name can now be a ZipInfo object.

ZipFile.testzip()
Read all the files in the archive and check their CRC's and file headers. Return the name of the first bad file, or else return None. Calling testzip() on a closed ZipFile will raise a RuntimeError.

ZipFile.write(filename[, arcname[, compress_type]])
Write the file named filename to the archive, giving it the archive name arcname (by default, this will be the same as filename, but without a drive letter and with leading path separators removed). If given, compress_type overrides the value given for the compression parameter to the constructor for the new entry. The archive must be open with mode 'w' or 'a' – calling write() on a ZipFile created with mode 'r' will raise a RuntimeError. Calling write() on a closed ZipFile will raise a RuntimeError.

Note: There is no official file name encoding for ZIP files. If you have unicode file names, you must

convert them to byte strings in your desired encoding before passing them to write(). WinZip interprets all file names as encoded in CP437, also known as DOS Latin.

Note: Archive names should be relative to the archive root, that is, they should not start with a path separator.

Note: If arcname (or filename, if arcname is not given) contains a null byte, the name of the file in the archive will be truncated at the null byte.

ZipFile.writestr(zinfo_or_arcname, bytes[, compress_type])
Write the string bytes to the archive; zinfo_or_arcname is either the file name it will be given in the archive, or a ZipInfo instance. If it's an instance, at least the filename, date, and time must be given. If it's a name, the date and time is set to the current date and time. The archive must be opened with mode `'w'` or `'a'` – calling writestr() on a ZipFile created with mode `'r'` will raise a RuntimeError. Calling writestr() on a closed ZipFile will raise a RuntimeError.

If given, compress_type overrides the value given for the compression parameter to the constructor for the new entry, or in the zinfo_or_arcname (if that is a ZipInfo instance).

Note: When passing a ZipInfo instance as the zinfo_or_arcname parameter, the compression method used will be that specified in the compress_type member of the given ZipInfo instance. By default, the ZipInfo constructor sets this member to ZIP_STORED.

Changed in version 2.7: The compress_type argument.

The following data attributes are also available:

ZipFile.debug
The level of debug output to use. This may be set from 0 (the default, no output) to 3 (the most output). Debugging information is written to sys.stdout.

ZipFile.comment
The comment text associated with the ZIP file. If assigning a comment to a ZipFile instance created with mode 'a' or 'w', this should be a string no longer than 65535 bytes. Comments longer than this will be truncated in the written archive when close() is called.

12.4.2 PyZipFile Objects

The PyZipFile constructor takes the same parameters as the ZipFile constructor. Instances have one method in addition to those of ZipFile objects.

PyZipFile.writepy(pathname[, basename])
Search for files *.py and add the corresponding file to the archive. The corresponding file is a *.pyo file if available, else a *.pyc file, compiling if necessary. If the pathname is a file, the filename must end with .py, and just the (corresponding *.py[co]) file is added at the top level (no path information). If the pathname is a file that does not end with .py, a RuntimeError will be raised. If it is a directory, and the directory is not a package directory, then all the files *.py[co] are added at the top level. If the directory is a package directory, then all *.py[co] are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively. basename is intended for internal use only. The writepy() method makes archives with file names like this:

```
string.pyc                      # Top level name
test/__init__.pyc               # Package directory
test/test_support.pyc           # Module test.test_support
test/bogus/__init__.pyc         # Subpackage directory
test/bogus/myfile.pyc           # Submodule test.bogus.myfile
```

12.4.3 ZipInfo Objects

Instances of the ZipInfo class are returned by the getinfo() and infolist() methods of ZipFile objects. Each object stores information about a single member of the ZIP archive.

Instances have the following attributes:

ZipInfo.filename
: Name of the file in the archive.

ZipInfo.date_time
: The time and date of the last modification to the archive member. This is a tuple of six values:

Index	Value
0	Year (>= 1980)
1	Month (one-based)
2	Day of month (one-based)
3	Hours (zero-based)
4	Minutes (zero-based)
5	Seconds (zero-based)

Note: The ZIP file format does not support timestamps before 1980.

ZipInfo.compress_type
: Type of compression for the archive member.

ZipInfo.comment
: Comment for the individual archive member.

ZipInfo.extra
: Expansion field data. The PKZIP Application Note contains some comments on the internal structure of the data contained in this string.

ZipInfo.create_system
: System which created ZIP archive.

ZipInfo.create_version
: PKZIP version which created ZIP archive.

ZipInfo.extract_version
: PKZIP version needed to extract archive.

ZipInfo.reserved
: Must be zero.

ZipInfo.flag_bits
: ZIP flag bits.

ZipInfo.volume
: Volume number of file header.

ZipInfo.internal_attr
: Internal attributes.

ZipInfo.external_attr
: External file attributes.

ZipInfo.header_offset
: Byte offset to the file header.

ZipInfo.CRC
: CRC-32 of the uncompressed file.

ZipInfo.compress_size
: Size of the compressed data.

ZipInfo.file_size
: Size of the uncompressed file.

12.4.4 Command-Line Interface

The zipfile module provides a simple command-line interface to interact with ZIP archives.

If you want to create a new ZIP archive, specify its name after the -c option and then list the filename(s) that should be included:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

Passing a directory is also acceptable:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

If you want to extract a ZIP archive into the specified directory, use the -e option:

```
$ python -m zipfile -e monty.zip target-dir/
```

For a list of the files in a ZIP archive, use the -l option:

```
$ python -m zipfile -l monty.zip
```

Command-line options

-l <zipfile>
: List files in a zipfile.

-c <zipfile> <source1> ... <sourceN>
: Create zipfile from source files.

-e <zipfile> <output_dir>
: Extract zipfile into target directory.

-t <zipfile>
: Test whether the zipfile is valid or not.

12.5 tarfile — Read and write tar archive files

New in version 2.3.

Source code: Lib/tarfile.py

The tarfile module makes it possible to read and write tar archives, including those using gzip or bz2 compression. Use the zipfile module to read or write .zip files, or the higher-level functions in shutil.

Some facts and figures:

- reads and writes gzip and bz2 compressed archives if the respective modules are available.
- read/write support for the POSIX.1-1988 (ustar) format.
- read/write support for the GNU tar format including longname and longlink extensions, read-only support for the sparse extension.
- read/write support for the POSIX.1-2001 (pax) format.

  New in version 2.6.

- handles directories, regular files, hardlinks, symbolic links, fifos, character devices and block devices and is able to acquire and restore file information like timestamp, access permissions and owner.

Note: Handling of multi-stream bzip2 files is not supported. Modules such as bz2file let you overcome this.

---

tarfile.open(name=None, mode='r', fileobj=None, bufsize=10240, **kwargs)

Return a TarFile object for the pathname name. For detailed information on TarFile objects and the keyword arguments that are allowed, see TarFile Objects.

mode has to be a string of the form `'filemode[:compression]'`, it defaults to `'r'`. Here is a full list of mode combinations:

mode	action
`'r'` or `'r:*'`	Open for reading with transparent compression (recommended).
`'r:'`	Open for reading exclusively without compression.
`'r:gz'`	Open for reading with gzip compression.
`'r:bz2'`	Open for reading with bzip2 compression.
`'a'` or `'a:'`	Open for appending with no compression. The file is created if it does not exist.
`'w'` or `'w:'`	Open for uncompressed writing.
`'w:gz'`	Open for gzip compressed writing.
`'w:bz2'`	Open for bzip2 compressed writing.

Note that `'a:gz'` or `'a:bz2'` is not possible. If mode is not suitable to open a certain (compressed) file for reading, ReadError is raised. Use mode `'r'` to avoid this. If a compression method is not supported, CompressionError is raised.

If fileobj is specified, it is used as an alternative to a file object opened for name. It is supposed to be at position 0.

For modes `'w:gz'`, `'r:gz'`, `'w:bz2'`, `'r:bz2'`, tarfile.open() accepts the keyword argument compresslevel (default 9) to specify the compression level of the file.

For special purposes, there is a second format for mode: `'filemode|[compression]'`. tarfile.open() will return a TarFile object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, fileobj may be any object that has a read() or write() method (depending on the mode). bufsize specifies the blocksize and defaults to 20 * 512 bytes. Use this variant in combination with e.g. sys.stdin, a socket file object or a tape device. However, such a TarFile object is limited in that it does not allow random access, see Examples. The currently possible modes:

Mode	Action
'r|*'	Open a stream of tar blocks for reading with transparent compression.
'r|'	Open a stream of uncompressed tar blocks for reading.
'r|gz'	Open a gzip compressed stream for reading.
'r|bz2'	Open a bzip2 compressed stream for reading.
'w|'	Open an uncompressed stream for writing.
'w|gz'	Open a gzip compressed stream for writing.
'w|bz2'	Open a bzip2 compressed stream for writing.

class tarfile.TarFile
Class for reading and writing tar archives. Do not use this class directly, better use tarfile.open() instead. See TarFile Objects.

tarfile.is_tarfile(name)
Return True if name is a tar archive file, that the tarfile module can read.

class tarfile.TarFileCompat(filename, mode='r', compression=TAR_PLAIN)
Class for limited access to tar archives with a zipfile-like interface. Please consult the documentation of the zipfile module for more details. compression must be one of the following constants:

TAR_PLAIN
Constant for an uncompressed tar archive.

TAR_GZIPPED
Constant for a gzip compressed tar archive.

Deprecated since version 2.6: The TarFileCompat class has been removed in Python 3.

exception tarfile.TarError
Base class for all tarfile exceptions.

exception tarfile.ReadError
Is raised when a tar archive is opened, that either cannot be handled by the tarfile module or is somehow invalid.

exception tarfile.CompressionError
Is raised when a compression method is not supported or when the data cannot be decoded properly.

exception tarfile.StreamError
Is raised for the limitations that are typical for stream-like TarFile objects.

exception tarfile.ExtractError
Is raised for non-fatal errors when using TarFile.extract(), but only if TarFile.errorlevel== 2.

The following constants are available at the module level:

tarfile.ENCODING
The default character encoding: 'utf-8' on Windows, the value returned by sys.getfilesystemencoding() otherwise.

exception tarfile.HeaderError
Is raised by TarInfo.frombuf() if the buffer it gets is invalid.

New in version 2.6.

Each of the following constants defines a tar archive format that the tarfile module is able to create. See section Supported tar formats for details.

tarfile.USTAR_FORMAT
POSIX.1-1988 (ustar) format.

tarfile.GNU_FORMAT
GNU tar format.

tarfile.PAX_FORMAT
POSIX.1-2001 (pax) format.

tarfile.DEFAULT_FORMAT
The default format for creating archives. This is currently GNU_FORMAT.

See also:

Module zipfile Documentation of the zipfile standard module.

Archiving operations Documentation of the higher-level archiving facilities provided by the standard shutil module.

GNU tar manual, Basic Tar Format Documentation for tar archive files, including GNU tar extensions.

12.5.1 TarFile Objects

The TarFile object provides an interface to a tar archive. A tar archive is a sequence of blocks. An archive member (a stored file) is made up of a header block followed by data blocks. It is possible to store a file in a tar archive several times. Each archive member is represented by a TarInfo object, see TarInfo Objects for details.

A TarFile object can be used as a context manager in a with statement. It will automatically be closed when the block is completed. Please note that in the event of an exception an archive opened for writing will not be finalized; only the internally used file object will be closed. See the Examples section for a use case.

New in version 2.7: Added support for the context management protocol.

class tarfile.TarFile(name=None, mode='r', fileobj=None, format=DEFAULT_FORMAT, tarinfo=TarInfo, dereference=False, ignore_zeros=False, encoding=ENCODING, errors=None, pax_headers=None, debug=0, errorlevel=0)
All following arguments are optional and can be accessed as instance attributes as well.

name is the pathname of the archive. It can be omitted if fileobj is given. In this case, the file object's name attribute is used if it exists.

mode is either `'r'` to read from an existing archive, `'a'` to append data to an existing file or `'w'` to create a new file overwriting an existing one.

If fileobj is given, it is used for reading or writing data. If it can be determined, mode is overridden by fileobj's mode. fileobj will be used from position 0.

Note: fileobj is not closed, when TarFile is closed.

---

format controls the archive format. It must be one of the constants USTAR_FORMAT, GNU_FORMAT or PAX_FORMAT that are defined at module level.

New in version 2.6.

The tarinfo argument can be used to replace the default TarInfo class with a different one.

New in version 2.6.

If dereference is False, add symbolic and hard links to the archive. If it is True, add the content of the target files to the archive. This has no effect on systems that do not support symbolic links.

If ignore_zeros is False, treat an empty block as the end of the archive. If it is True, skip empty (and invalid) blocks and try to get as many members as possible. This is only useful for reading concatenated or damaged archives.

debug can be set from 0 (no debug messages) up to 3 (all debug messages). The messages are written to sys.stderr.

If errorlevel is 0, all errors are ignored when using TarFile.extract(). Nevertheless, they appear as error messages in the debug output, when debugging is enabled. If 1, all fatal errors are raised as OSError or IOError exceptions. If 2, all non-fatal errors are raised as TarError exceptions as well.

The encoding and errors arguments control the way strings are converted to unicode objects and vice versa. The default settings will work for most users. See section Unicode issues for in-depth information.

New in version 2.6.

The pax_headers argument is an optional dictionary of unicode strings which will be added as a pax global header if format is PAX_FORMAT.

New in version 2.6.

classmethod TarFile.open(...)
Alternative constructor. The tarfile.open() function is actually a shortcut to this classmethod.

TarFile.getmember(name)
Return a TarInfo object for member name. If name can not be found in the archive, KeyError is raised.

Note: If a member occurs more than once in the archive, its last occurrence is assumed to be the most up-to-date version.

TarFile.getmembers()
Return the members of the archive as a list of TarInfo objects. The list has the same order as the members in the archive.

TarFile.getnames()
Return the members as a list of their names. It has the same order as the list returned by getmembers().

TarFile.list(verbose=True)
Print a table of contents to sys.stdout. If verbose is False, only the names of the members are printed. If it is True, output similar to that of ls -l is produced.

TarFile.next()
Return the next member of the archive as a TarInfo object, when TarFile is opened for reading. Return None if there is no more available.

TarFile.extractall(path=".", members=None)
Extract all members from the archive to the current working directory or directory path. If optional members is given, it must be a subset of the list returned by getmembers(). Directory information like owner, modification time and permissions are set after all members have been extracted. This is done to work around two problems: A directory's modification time is reset each time a file is created in it. And, if a directory's permissions do not allow writing, extracting files to it will fail.

Warning: Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of path, e.g. members that have absolute filenames starting with "/" or filenames with two dots "..".

New in version 2.5.

TarFile.extract(member, path="")
Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. member may be a filename or a TarInfo object. You can specify a different directory using path.

Note: The extract() method does not take care of several extraction issues. In most cases you should consider using the extractall() method.

Warning: See the warning for extractall().

TarFile.extractfile(member)
Extract a member from the archive as a file object. member may be a filename or a TarInfo object. If member is a regular file, a file-like object is returned. If member is a link, a file-like object is constructed from the link's target. If member is none of the above, None is returned.

Note: The file-like object is read-only. It provides the methods read(), readline(), readlines(), seek(), tell(), and close(), and also supports iteration over its lines.

TarFile.add(name, arcname=None, recursive=True, exclude=None, filter=None)
Add the file name to the archive. name may be any type of file (directory, fifo, symbolic link, etc.). If given, arcname specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting recursive to False. If exclude is given it must be a function that takes one filename argument and returns a boolean value. Depending on this value the respective file is either excluded (True) or added (False). If filter is specified it must be a function that takes a TarInfo object argument and returns the changed TarInfo object. If it instead returns None the TarInfo object will be excluded from the archive. See Examples for an example.

Changed in version 2.6: Added the exclude parameter.

Changed in version 2.7: Added the filter parameter.

Deprecated since version 2.7: The exclude parameter is deprecated, please use the filter parameter instead. For maximum portability, filter should be used as a keyword argument rather than as a positional argument so that code won't be affected when exclude is ultimately removed.

TarFile.addfile(tarinfo, fileobj=None)
Add the TarInfo object tarinfo to the archive. If fileobj is given, tarinfo.size bytes are read from it and added to the archive. You can create TarInfo objects directly, or by using gettarinfo().

Note: On Windows platforms, fileobj should always be opened with mode 'rb' to avoid irritation about the file size.

TarFile.gettarinfo(name=None, arcname=None, fileobj=None)
Create a TarInfo object from the result of os.stat() or equivalent on an existing file. The file is either named by name, or specified as a file object fileobj with a file descriptor. If given, arcname specifies an alternative name for the file in the archive, otherwise, the name is taken from fileobj's name attribute, or the name argument.

You can modify some of the TarInfo's attributes before you add it using addfile(). If the file object is not an ordinary file object positioned at the beginning of the file, attributes such as size may need

modifying. This is the case for objects such as GzipFile. The name may also be modified, in which case arcname could be a dummy string.

TarFile.close()
Close the TarFile. In write mode, two finishing zero blocks are appended to the archive.

TarFile.posix
Setting this to True is equivalent to setting the format attribute to USTAR_FORMAT, False is equivalent to GNU_FORMAT.

Changed in version 2.4: posix defaults to False.

Deprecated since version 2.6: Use the format attribute instead.

TarFile.pax_headers
A dictionary containing key-value pairs of pax global headers.

New in version 2.6.

12.5.2 TarInfo Objects

A TarInfo object represents one member in a TarFile. Aside from storing all required attributes of a file (like file type, size, time, permissions, owner etc.), it provides some useful methods to determine its type. It does not contain the file's data itself.

TarInfo objects are returned by TarFile's methods getmember(), getmembers() and gettarinfo().

class tarfile.TarInfo(name="")
Create a TarInfo object.

TarInfo.frombuf(buf)
Create and return a TarInfo object from string buffer buf.

New in version 2.6: Raises HeaderError if the buffer is invalid..

TarInfo.fromtarfile(tarfile)
Read the next member from the TarFile object tarfile and return it as a TarInfo object.

New in version 2.6.

TarInfo.tobuf(format=DEFAULT_FORMAT, encoding=ENCODING, errors='strict')
Create a string buffer from a TarInfo object. For information on the arguments see the constructor of the TarFile class.

Changed in version 2.6: The arguments were added.

A TarInfo object has the following public data attributes:

TarInfo.name
Name of the archive member.

TarInfo.size
Size in bytes.

TarInfo.mtime
Time of last modification.

TarInfo.mode
Permission bits.

TarInfo.type
File type. type is usually one of these constants: REGTYPE, AREGTYPE, LNKTYPE, SYMTYPE,

DIRTYPE, FIFOTYPE, CONTTYPE, CHRTYPE, BLKTYPE, GNUTYPE_SPARSE. To determine the type of a TarInfo object more conveniently, use the is*() methods below.

TarInfo.linkname
Name of the target file name, which is only present in TarInfo objects of type LNKTYPE and SYMTYPE.

TarInfo.uid
User ID of the user who originally stored this member.

TarInfo.gid
Group ID of the user who originally stored this member.

TarInfo.uname
User name.

TarInfo.gname
Group name.

TarInfo.pax_headers
A dictionary containing key-value pairs of an associated pax extended header.

New in version 2.6.

A TarInfo object also provides some convenient query methods:

TarInfo.isfile()
Return True if the Tarinfo object is a regular file.

TarInfo.isreg()
Same as isfile().

TarInfo.isdir()
Return True if it is a directory.

TarInfo.issym()
Return True if it is a symbolic link.

TarInfo.islnk()
Return True if it is a hard link.

TarInfo.ischr()
Return True if it is a character device.

TarInfo.isblk()
Return True if it is a block device.

TarInfo.isfifo()
Return True if it is a FIFO.

TarInfo.isdev()
Return True if it is one of character device, block device or FIFO.

12.5.3 Examples

How to extract an entire tar archive to the current working directory:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

How to extract a subset of a tar archive with TarFile.extractall() using a generator function instead of a list:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

How to create an uncompressed tar archive from a list of filenames:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

The same example using the with statement:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

How to read a gzip compressed tar archive and display some member information:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print tarinfo.name, "is", tarinfo.size, "bytes in size and is",
    if tarinfo.isreg():
        print "a regular file."
    elif tarinfo.isdir():
        print "a directory."
    else:
        print "something else."
tar.close()
```

How to create an archive and reset the user information using the filter parameter in TarFile.add():

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

12.5.4 Supported tar formats

There are three tar formats that can be created with the tarfile module:

- The POSIX.1-1988 ustar format (USTAR_FORMAT). It supports filenames up to a length of at best 256 characters and linknames up to 100 characters. The maximum file size is 8 gigabytes. This is an old and limited but widely supported format.
- The GNU tar format (GNU_FORMAT). It supports long filenames and linknames, files bigger than 8 gigabytes and sparse files. It is the de facto standard on GNU/Linux systems. tarfile fully supports the GNU tar extensions for long names, sparse file support is read-only.
- The POSIX.1-2001 pax format (PAX_FORMAT). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. However, not all tar implementations today are able to handle pax archives properly.

  The pax format is an extension to the existing ustar format. It uses extra headers for information that cannot be stored otherwise. There are two flavours of pax headers: Extended headers only affect the subsequent file header, global headers are valid for the complete archive and affect all following files. All the data in a pax header is encoded in UTF-8 for portability reasons.

There are some more variants of the tar format which can be read, but not created:

- The ancient V7 format. This is the first tar format from Unix Seventh Edition, storing only regular files and directories. Names must not be longer than 100 characters, there is no user/group name information. Some archives have miscalculated header checksums in case of fields with non-ASCII characters.
- The SunOS tar extended format. This format is a variant of the POSIX.1-2001 pax format, but is not compatible.

12.5.5 Unicode issues

The tar format was originally conceived to make backups on tape drives with the main focus on preserving file system information. Nowadays tar archives are commonly used for file distribution and exchanging archives over networks. One problem of the original format (that all other formats are merely variants of) is that there is no concept of supporting different character encodings. For example, an ordinary tar archive created on a UTF-8 system cannot be read correctly on a Latin-1 system if it contains non-ASCII characters. Names (i.e. filenames, linknames, user/group names) containing these characters will appear damaged. Unfortunately, there is no way to autodetect the encoding of an archive.

The pax format was designed to solve this problem. It stores non-ASCII names using the universal character encoding UTF-8. When a pax archive is read, these UTF-8 names are converted to the encoding of the local file system.

The details of unicode conversion are controlled by the encoding and errors keyword arguments of the TarFile class.

The default value for encoding is the local character encoding. It is deduced from sys.getfilesystemencoding() and sys.getdefaultencoding(). In read mode, encoding is used exclusively to convert unicode names from a pax archive to strings in the local character encoding. In write mode, the use of encoding depends on the chosen archive format. In case of PAX_FORMAT, input names that contain non-ASCII characters need to be decoded before being stored as UTF-8 strings. The other formats do not make use of encoding unless unicode objects are used as input names. These are converted to 8-bit character strings before they are added to the archive.

The errors argument defines how characters are treated that cannot be converted to or from encoding. Possible values are listed in section Codec Base Classes. In read mode, there is an additional scheme `'utf-8'` which means that bad characters are replaced by their UTF-8 representation. This is the default scheme. In write mode the default value for errors is `'strict'` to ensure that name information is not altered unnoticed.

CHAPTER

THIRTEEN

FILE FORMATS

The modules described in this chapter parse various miscellaneous file formats that aren't markup languages or are related to e-mail.

13.1 csv — CSV File Reading and Writing

New in version 2.3.

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. There is no "CSV standard", so the format is operationally defined by the many applications which read and write it. The lack of a standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The csv module implements classes to read and write tabular data in CSV format. It allows programmers to say, "write this data in the format preferred by Excel," or "read data from this file which was generated by Excel," without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

The csv module's reader and writer objects read and write sequences. Programmers can also read and write data in dictionary form using the DictReader and DictWriter classes.

Note: This version of the csv module doesn't support Unicode input. Also, there are currently some issues regarding ASCII NUL characters. Accordingly, all input should be UTF-8 or printable ASCII to be safe; see the examples in section Examples.

See also:

PEP 305 - CSV File API The Python Enhancement Proposal which proposed this addition to Python.

13.1.1 Module Contents

The csv module defines the following functions:

csv.reader(csvfile, dialect='excel', **fmtparams)

Return a reader object which will iterate over lines in the given csvfile. csvfile can be any object which supports the iterator protocol and returns a string each time its next() method is called — file objects and list objects are both suitable. If csvfile is a file object, it must be opened with the 'b' flag on platforms where that makes a difference. An optional dialect parameter can be given which

is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the Dialect class or one of the strings returned by the list_dialects() function. The other optional fmtparams keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters.

Each row read from the csv file is returned as a list of strings. No automatic data type conversion is performed.

A short usage example:

```
>>> import csv
>>> with open('eggs.csv', 'rb') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print ', '.join(row)
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

Changed in version 2.5: The parser is now stricter with respect to multi-line quoted fields. Previously, if a line ended within a quoted field without a terminating newline character, a newline would be inserted into the returned field. This behavior caused problems when reading files which contained carriage return characters within fields. The behavior was changed to return the field without inserting newlines. As a consequence, if newlines embedded within fields are important, the input should be split into lines in a manner which preserves the newline characters.

csv.writer(csvfile, dialect='excel', **fmtparams)

Return a writer object responsible for converting the user's data into delimited strings on the given file-like object. csvfile can be any object with a write() method. If csvfile is a file object, it must be opened with the 'b' flag on platforms where that makes a difference. An optional dialect parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the Dialect class or one of the strings returned by the list_dialects() function. The other optional fmtparams keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters. To make it as easy as possible to interface with modules which implement the DB API, the value None is written as the empty string. While this isn't a reversible transformation, it makes it easier to dump SQL NULL data values to CSV files without preprocessing the data returned from a cursor.fetch* call. Floats are stringified with repr() before being written. All other non-string data are stringified with str() before being written.

A short usage example:

```
import csv
with open('eggs.csv', 'wb') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

csv.register_dialect(name[, dialect], **fmtparams)

Associate dialect with name. name must be a string or Unicode object. The dialect can be specified either by passing a sub-class of Dialect, or by fmtparams keyword arguments, or both, with keyword arguments overriding parameters of the dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters.

csv.unregister_dialect(name)

Delete the dialect associated with name from the dialect registry. An Error is raised if name is not a registered dialect name.

csv.get_dialect(name)
Return the dialect associated with name. An Error is raised if name is not a registered dialect name.

Changed in version 2.5: This function now returns an immutable Dialect. Previously an instance of the requested dialect was returned. Users could modify the underlying class, changing the behavior of active readers and writers.

csv.list_dialects()
Return the names of all registered dialects.

csv.field_size_limit([new_limit])
Returns the current maximum field size allowed by the parser. If new_limit is given, this becomes the new limit.

New in version 2.5.

The csv module defines the following classes:

class csv.DictReader(f, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwds)
Create an object which operates like a regular reader but maps the information read into a dict whose keys are given by the optional fieldnames parameter. The fieldnames parameter is a sequence whose elements are associated with the fields of the input data in order. These elements become the keys of the resulting dictionary. If the fieldnames parameter is omitted, the values in the first row of the file f will be used as the fieldnames. If the row read has more fields than the fieldnames sequence, the remaining data is added as a sequence keyed by the value of restkey. If the row read has fewer fields than the fieldnames sequence, the remaining keys take the value of the optional restval parameter. Any other optional or keyword arguments are passed to the underlying reader instance.

A short usage example:

```
>>> import csv
>>> with open('names.csv') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Baked Beans
Lovely Spam
Wonderful Spam
```

class csv.DictWriter(f, fieldnames, restval='', extrasaction='raise', dialect='excel', *args, **kwds)
Create an object which operates like a regular writer but maps dictionaries onto output rows. The fieldnames parameter is a sequence of keys that identify the order in which values in the dictionary passed to the writerow() method are written to the file f. The optional restval parameter specifies the value to be written if the dictionary is missing a key in fieldnames. If the dictionary passed to the writerow() method contains a key not found in fieldnames, the optional extrasaction parameter indicates what action to take. If it is set to 'raise' a ValueError is raised. If it is set to 'ignore', extra values in the dictionary are ignored. Any other optional or keyword arguments are passed to the underlying writer instance.

Note that unlike the DictReader class, the fieldnames parameter of the DictWriter is not optional. Since Python's dict objects are not ordered, there is not enough information available to deduce the order in which the row should be written to the file f.

A short usage example:

```
import csv
```

(continues on next page)

(continued from previous page)

```
with open('names.csv', 'w') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class csv.Dialect
: The Dialect class is a container class relied on primarily for its attributes, which are used to define the parameters for a specific reader or writer instance.

class csv.excel
: The excel class defines the usual properties of an Excel-generated CSV file. It is registered with the dialect name 'excel'.

class csv.excel_tab
: The excel_tab class defines the usual properties of an Excel-generated TAB-delimited file. It is registered with the dialect name 'excel-tab'.

class csv.Sniffer
: The Sniffer class is used to deduce the format of a CSV file.

  The Sniffer class provides two methods:

  sniff(sample, delimiters=None)
  : Analyze the given sample and return a Dialect subclass reflecting the parameters found. If the optional delimiters parameter is given, it is interpreted as a string containing possible valid delimiter characters.

  has_header(sample)
  : Analyze the sample text (presumed to be in CSV format) and return True if the first row appears to be a series of column headers.

An example for Sniffer use:

```
with open('example.csv', 'rb') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

The csv module defines the following constants:

csv.QUOTE_ALL
: Instructs writer objects to quote all fields.

csv.QUOTE_MINIMAL
: Instructs writer objects to only quote those fields which contain special characters such as delimiter, quotechar or any of the characters in lineterminator.

csv.QUOTE_NONNUMERIC
: Instructs writer objects to quote all non-numeric fields.

  Instructs the reader to convert all non-quoted fields to type float.

csv.QUOTE_NONE
: Instructs writer objects to never quote fields. When the current delimiter occurs in output data it is

preceded by the current escapechar character. If escapechar is not set, the writer will raise Error if any characters that require escaping are encountered.

Instructs reader to perform no special processing of quote characters.

The csv module defines the following exception:

exception csv.Error
Raised by any of the functions when an error is detected.

13.1.2 Dialects and Formatting Parameters

To make it easier to specify the format of input and output records, specific formatting parameters are grouped together into dialects. A dialect is a subclass of the Dialect class having a set of specific methods and a single validate() method. When creating reader or writer objects, the programmer can specify a string or a subclass of the Dialect class as the dialect parameter. In addition to, or instead of, the dialect parameter, the programmer can also specify individual formatting parameters, which have the same names as the attributes defined below for the Dialect class.

Dialects support the following attributes:

Dialect.delimiter
A one-character string used to separate fields. It defaults to `','`.

Dialect.doublequote
Controls how instances of quotechar appearing inside a field should themselves be quoted. When True, the character is doubled. When False, the escapechar is used as a prefix to the quotechar. It defaults to True.

On output, if doublequote is False and no escapechar is set, Error is raised if a quotechar is found in a field.

Dialect.escapechar
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False. On reading, the escapechar removes any special meaning from the following character. It defaults to None, which disables escaping.

Dialect.lineterminator
The string used to terminate lines produced by the writer. It defaults to `'\r\n'`.

Note: The reader is hard-coded to recognise either `'\r'` or `'\n'` as end-of-line, and ignores lineterminator. This behavior may change in the future.

Dialect.quotechar
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to `'"'`.

Dialect.quoting
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

Dialect.skipinitialspace
When True, whitespace immediately following the delimiter is ignored. The default is False.

Dialect.strict
When True, raise exception Error on bad CSV input. The default is False.

13.1.3 Reader Objects

Reader objects (DictReader instances and objects returned by the reader() function) have the following public methods:

csvreader.next()
: Return the next row of the reader's iterable object as a list, parsed according to the current dialect.

Reader objects have the following public attributes:

csvreader.dialect
: A read-only description of the dialect in use by the parser.

csvreader.line_num
: The number of lines read from the source iterator. This is not the same as the number of records returned, as records can span multiple lines.

  New in version 2.5.

DictReader objects have the following public attribute:

csvreader.fieldnames
: If not passed as a parameter when creating the object, this attribute is initialized upon first access or when the first record is read from the file.

  Changed in version 2.6.

13.1.4 Writer Objects

Writer objects (DictWriter instances and objects returned by the writer() function) have the following public methods. A row must be a sequence of strings or numbers for Writer objects and a dictionary mapping fieldnames to strings or numbers (by passing them through str() first) for DictWriter objects. Note that complex numbers are written out surrounded by parens. This may cause some problems for other programs which read CSV files (assuming they support complex numbers at all).

csvwriter.writerow(row)
: Write the row parameter to the writer's file object, formatted according to the current dialect.

csvwriter.writerows(rows)
: Write all elements in rows (an iterable of row objects as described above) to the writer's file object, formatted according to the current dialect.

Writer objects have the following public attribute:

csvwriter.dialect
: A read-only description of the dialect in use by the writer.

DictWriter objects have the following public method:

DictWriter.writeheader()
: Write a row with the field names (as specified in the constructor).

  New in version 2.7.

13.1.5 Examples

The simplest example of reading a CSV file:

```
import csv
with open('some.csv', 'rb') as f:
    reader = csv.reader(f)
    for row in reader:
        print row
```

Reading a file with an alternate format:

```
import csv
with open('passwd', 'rb') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print row
```

The corresponding simplest possible writing example is:

```
import csv
with open('some.csv', 'wb') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

Registering a new dialect:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', 'rb') as f:
    reader = csv.reader(f, 'unixpwd')
```

A slightly more advanced use of the reader — catching and reporting errors:

```
import csv, sys
filename = 'some.csv'
with open(filename, 'rb') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print row
    except csv.Error as e:
        sys.exit('file %s, line %d: %s' % (filename, reader.line_num, e))
```

And while the module doesn't directly support parsing strings, it can easily be done:

```
import csv
for row in csv.reader(['one,two,three']):
    print row
```

The csv module doesn't directly support reading and writing Unicode, but it is 8-bit-clean save for some problems with ASCII NUL characters. So you can write functions or classes that handle the encoding and decoding for you as long as you avoid encodings like UTF-16 that use NULs. UTF-8 is recommended.

unicode_csv_reader() below is a generator that wraps csv.reader to handle Unicode CSV data (a list of Unicode strings). utf_8_encoder() is a generator that encodes the Unicode strings as UTF-8, one string (or row) at a time. The encoded strings are parsed by the CSV reader, and unicode_csv_reader() decodes the UTF-8-encoded cells back into Unicode:

```
import csv
```

(continues on next page)

(continued from previous page)

```
def unicode_csv_reader(unicode_csv_data, dialect=csv.excel, **kwargs):
    # csv.py doesn't do Unicode; encode temporarily as UTF-8:
    csv_reader = csv.reader(utf_8_encoder(unicode_csv_data),
                            dialect=dialect, **kwargs)
    for row in csv_reader:
        # decode UTF-8 back to Unicode, cell by cell:
        yield [unicode(cell, 'utf-8') for cell in row]

def utf_8_encoder(unicode_csv_data):
    for line in unicode_csv_data:
        yield line.encode('utf-8')
```

For all other encodings the following UnicodeReader and UnicodeWriter classes can be used. They take an additional encoding parameter in their constructor and make sure that the data passes the real reader or writer encoded as UTF-8:

```
import csv, codecs, cStringIO

class UTF8Recoder:
    """
    Iterator that reads an encoded stream and reencodes the input to UTF-8
    """
    def __init__(self, f, encoding):
        self.reader = codecs.getreader(encoding)(f)

    def __iter__(self):
        return self

    def next(self):
        return self.reader.next().encode("utf-8")

class UnicodeReader:
    """
    A CSV reader which will iterate over lines in the CSV file "f",
    which is encoded in the given encoding.
    """

    def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwds):
        f = UTF8Recoder(f, encoding)
        self.reader = csv.reader(f, dialect=dialect, **kwds)

    def next(self):
        row = self.reader.next()
        return [unicode(s, "utf-8") for s in row]

    def __iter__(self):
        return self

class UnicodeWriter:
    """
    A CSV writer which will write rows to CSV file "f",
    which is encoded in the given encoding.
    """

    def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwds):
        # Redirect output to a queue
```

(continues on next page)

(continued from previous page)

```
        self.queue = cStringIO.StringIO()
        self.writer = csv.writer(self.queue, dialect=dialect, **kwds)
        self.stream = f
        self.encoder = codecs.getincrementalencoder(encoding)()

    def writerow(self, row):
        self.writer.writerow([s.encode("utf-8") for s in row])
        # Fetch UTF-8 output from the queue ...
        data = self.queue.getvalue()
        data = data.decode("utf-8")
        # ... and reencode it into the target encoding
        data = self.encoder.encode(data)
        # write to the target stream
        self.stream.write(data)
        # empty queue
        self.queue.truncate(0)

    def writerows(self, rows):
        for row in rows:
            self.writerow(row)
```

13.2 ConfigParser — Configuration file parser

Note: The ConfigParser module has been renamed to configparser in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

This module defines the class ConfigParser. The ConfigParser class implements a basic configuration file parser language which provides a structure similar to what you would find on Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

Note: This library does not interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

See also:

Module shlex Support for creating Unix shell-like mini-languages which can be used as an alternate format for application configuration files.

Module json The json module implements a subset of JavaScript syntax which can also be used for this purpose.

The configuration file consists of sections, led by a [section] header and followed by name: value entries, with continuations in the style of RFC 822 (see section 3.1.1, "LONG HEADER FIELDS"); name=value is also accepted. Note that leading whitespace is removed from values. The optional values can contain format strings which refer to other values in the same section, or values in a special DEFAULT section. Additional defaults can be provided on initialization and retrieval. Lines beginning with `'#'` or `';'` are ignored and may be used to provide comments.

Configuration files may include comments, prefixed by specific characters (# and ;). Comments may appear on their own in an otherwise empty line, or may be entered in lines holding values or section names. In the latter case, they need to be preceded by a whitespace character to be recognized as a comment. (For backwards compatibility, only ; starts an inline comment, while # does not.)

On top of the core functionality, SafeConfigParser supports interpolation. This means values can contain format strings which refer to other values in the same section, or values in a special DEFAULT section. Additional defaults can be provided on initialization.

For example:

```
[My Section]
foodir: %(dir)s/whatever
dir=frob
long: this value continues
   in the next line
```

would resolve the %(dir)s to the value of dir (frob in this case). All reference expansions are done on demand.

Default values can be specified by passing them into the ConfigParser constructor as a dictionary. Additional defaults may be passed into the get() method which will override all others.

Sections are normally stored in a built-in dictionary. An alternative dictionary type can be passed to the ConfigParser constructor. For example, if a dictionary type is passed that sorts its keys, the sections will be sorted on write-back, as will be the keys within each section.

class ConfigParser.RawConfigParser([defaults[, dict_type[, allow_no_value]]])
The basic configuration object. When defaults is given, it is initialized into the dictionary of intrinsic defaults. When dict_type is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values. When allow_no_value is true (default: False), options without values are accepted; the value presented for these is None.

This class does not support the magical interpolation behavior.

All option names are passed through the optionxform() method. Its default implementation converts option names to lower case.

New in version 2.3.

Changed in version 2.6: dict_type was added.

Changed in version 2.7: The default dict_type is collections.OrderedDict. allow_no_value was added.

class ConfigParser.ConfigParser([defaults[, dict_type[, allow_no_value]]])
Derived class of RawConfigParser that implements the magical interpolation feature and adds optional arguments to the get() and items() methods. The values in defaults must be appropriate for the %()s string interpolation. Note that __name__ is an intrinsic default; its value is the section name, and will override any value provided in defaults.

All option names used in interpolation will be passed through the optionxform() method just like any other option name reference. Using the default implementation of optionxform(), the values foo %(bar)s and foo %(BAR)s are equivalent.

New in version 2.3.

Changed in version 2.6: dict_type was added.

Changed in version 2.7: The default dict_type is collections.OrderedDict. allow_no_value was added.

class ConfigParser.SafeConfigParser([defaults[, dict_type[, allow_no_value]]])
Derived class of ConfigParser that implements a more-sane variant of the magical interpolation feature. This implementation is more predictable as well. New applications should prefer this version if they don't need to be compatible with older versions of Python.

New in version 2.3.

Changed in version 2.6: dict_type was added.

Changed in version 2.7: The default dict_type is collections.OrderedDict. allow_no_value was added.

exception ConfigParser.Error
Base class for all other configparser exceptions.

exception ConfigParser.NoSectionError
Exception raised when a specified section is not found.

exception ConfigParser.DuplicateSectionError
Exception raised if add_section() is called with the name of a section that is already present.

exception ConfigParser.NoOptionError
Exception raised when a specified option is not found in the specified section.

exception ConfigParser.InterpolationError
Base class for exceptions raised when problems occur performing string interpolation.

exception ConfigParser.InterpolationDepthError
Exception raised when string interpolation cannot be completed because the number of iterations exceeds MAX_INTERPOLATION_DEPTH. Subclass of InterpolationError.

exception ConfigParser.InterpolationMissingOptionError
Exception raised when an option referenced from a value does not exist. Subclass of InterpolationError.

New in version 2.3.

exception ConfigParser.InterpolationSyntaxError
Exception raised when the source text into which substitutions are made does not conform to the required syntax. Subclass of InterpolationError.

New in version 2.3.

exception ConfigParser.MissingSectionHeaderError
Exception raised when attempting to parse a file which has no section headers.

exception ConfigParser.ParsingError
Exception raised when errors occur attempting to parse a file.

ConfigParser.MAX_INTERPOLATION_DEPTH
The maximum depth for recursive interpolation for get() when the raw parameter is false. This is relevant only for the ConfigParser class.

See also:

Module shlex Support for a creating Unix shell-like mini-languages which can be used as an alternate format for application configuration files.

13.2.1 RawConfigParser Objects

RawConfigParser instances have the following methods:

RawConfigParser.defaults()
Return a dictionary containing the instance-wide defaults.

RawConfigParser.sections()
Return a list of the sections available; DEFAULT is not included in the list.

RawConfigParser.add_section(section)
Add a section named section to the instance. If a section by the given name already exists, DuplicateSectionError is raised. If the name DEFAULT (or any of it's case-insensitive variants) is passed, ValueError is raised.

RawConfigParser.has_section(section)
Indicates whether the named section is present in the configuration. The DEFAULT section is not acknowledged.

RawConfigParser.options(section)
Returns a list of options available in the specified section.

RawConfigParser.has_option(section, option)
If the given section exists, and contains the given option, return True; otherwise return False.

New in version 1.6.

RawConfigParser.read(filenames)
Attempt to read and parse a list of filenames, returning a list of filenames which were successfully parsed. If filenames is a string or Unicode string, it is treated as a single filename. If a file named in filenames cannot be opened, that file will be ignored. This is designed so that you can specify a list of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the list will be read. If none of the named files exist, the ConfigParser instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using readfp() before calling read() for any optional files:

```
import ConfigParser, os

config = ConfigParser.ConfigParser()
config.readfp(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')])
```

Changed in version 2.4: Returns list of successfully parsed filenames.

RawConfigParser.readfp(fp[, filename])
Read and parse configuration data from the file or file-like object in fp (only the readline() method is used). If filename is omitted and fp has a name attribute, that is used for filename; the default is <???>.

RawConfigParser.get(section, option)
Get an option value for the named section.

RawConfigParser.getint(section, option)
A convenience method which coerces the option in the specified section to an integer.

RawConfigParser.getfloat(section, option)
A convenience method which coerces the option in the specified section to a floating point number.

RawConfigParser.getboolean(section, option)
A convenience method which coerces the option in the specified section to a Boolean value. Note that the accepted values for the option are "1", "yes", "true", and "on", which cause this method to return True, and "0", "no", "false", and "off", which cause it to return False. These string values are checked in a case-insensitive manner. Any other value will cause it to raise ValueError.

RawConfigParser.items(section)
Return a list of (name, value) pairs for each option in the given section.

RawConfigParser.set(section, option, value)
If the given section exists, set the given option to the specified value; otherwise raise NoSectionError. While it is possible to use RawConfigParser (or ConfigParser with raw parameters set to true) for internal storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

New in version 1.6.

RawConfigParser.write(fileobject)
Write a representation of the configuration to the specified file object. This representation can be parsed by a future read() call.

New in version 1.6.

RawConfigParser.remove_option(section, option)
Remove the specified option from the specified section. If the section does not exist, raise NoSectionError. If the option existed to be removed, return True; otherwise return False.

New in version 1.6.

RawConfigParser.remove_section(section)
Remove the specified section from the configuration. If the section in fact existed, return True. Otherwise return False.

RawConfigParser.optionxform(option)
Transforms the option name option as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of option; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior.

You don't necessarily need to subclass a ConfigParser to use this method, you can also re-set it on an instance, to a function that takes a string argument. Setting it to str, for example, would make option names case sensitive:

```
cfgparser = ConfigParser()
...
cfgparser.optionxform = str
```

Note that when reading configuration files, whitespace around the option names are stripped before optionxform() is called.

13.2.2 ConfigParser Objects

The ConfigParser class extends some methods of the RawConfigParser interface, adding some optional arguments.

ConfigParser.get(section, option[, raw[, vars]])
Get an option value for the named section. If vars is provided, it must be a dictionary. The option is looked up in vars (if provided), section, and in defaults in that order.

All the '%' interpolations are expanded in the return values, unless the raw argument is true. Values for interpolation keys are looked up in the same manner as the option.

ConfigParser.items(section[, raw[, vars]])
Return a list of (name, value) pairs for each option in the given section. Optional arguments have the same meaning as for the get() method.

New in version 2.3.

13.2.3 SafeConfigParser Objects

The SafeConfigParser class implements the same extended interface as ConfigParser, with the following addition:

SafeConfigParser.set(section, option, value)
If the given section exists, set the given option to the specified value; otherwise raise NoSectionError. value must be a string (str or unicode); if not, TypeError is raised.

New in version 2.4.

13.2.4 Examples

An example of writing to a configuration file:

```
import ConfigParser

config = ConfigParser.RawConfigParser()

# When adding sections or items, add them in the reverse order of
# how you want them to be displayed in the actual file.
# In addition, please note that using RawConfigParser's and the raw
# mode of ConfigParser's respective set functions, you can assign
# non-string values to keys internally, but will receive an error
# when attempting to write to a file or when you get it in non-raw
# mode. SafeConfigParser does not allow such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'wb') as configfile:
    config.write(configfile)
```

An example of reading the configuration file again:

```
import ConfigParser

config = ConfigParser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print a_float + an_int

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print config.get('Section1', 'foo')
```

To get interpolation, you will need to use a ConfigParser or SafeConfigParser:

```
import ConfigParser

config = ConfigParser.ConfigParser()
config.read('example.cfg')
```

(continues on next page)

(continued from previous page)

```
# Set the third, optional argument of get to 1 if you wish to use raw mode.
print config.get('Section1', 'foo', 0)  # -> "Python is fun!"
print config.get('Section1', 'foo', 1)  # -> "%(bar)s is %(baz)s!"

# The optional fourth argument is a dict with members that will take
# precedence in interpolation.
print config.get('Section1', 'foo', 0, {'bar': 'Documentation',
                                        'baz': 'evil'})
```

Defaults are available in all three types of ConfigParsers. They are used in interpolation if an option used is not defined elsewhere.

```
import ConfigParser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = ConfigParser.SafeConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print config.get('Section1', 'foo')  # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print config.get('Section1', 'foo')  # -> "Life is hard!"
```

The function opt_move below can be used to move options between sections:

```
def opt_move(config, section1, section2, option):
    try:
        config.set(section2, option, config.get(section1, option, 1))
    except ConfigParser.NoSectionError:
        # Create non-existent section
        config.add_section(section2)
        opt_move(config, section1, section2, option)
    else:
        config.remove_option(section1, option)
```

Some configuration files are known to include settings without values, but which otherwise conform to the syntax supported by ConfigParser. The allow_no_value parameter to the constructor can be used to indicate that such values should be accepted:

```
>>> import ConfigParser
>>> import io

>>> sample_config = """
... [mysqld]
... user = mysql
... pid-file = /var/run/mysqld/mysqld.pid
... skip-external-locking
... old_passwords = 1
... skip-bdb
... skip-innodb
... """
>>> config = ConfigParser.RawConfigParser(allow_no_value=True)
>>> config.readfp(io.BytesIO(sample_config))

>>> # Settings with values are treated as before:
```

(continues on next page)

(continued from previous page)

```
>>> config.get("mysqld", "user")
'mysql'

>>> # Settings without values provide None:
>>> config.get("mysqld", "skip-bdb")

>>> # Settings which aren't specified still raise an error:
>>> config.get("mysqld", "does-not-exist")
Traceback (most recent call last):
  ...
ConfigParser.NoOptionError: No option 'does-not-exist' in section: 'mysqld'
```

13.3 robotparser — Parser for robots.txt

Note: The robotparser module has been renamed urllib.robotparser in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

This module provides a single class, RobotFileParser, which answers questions about whether or not a particular user agent can fetch a URL on the Web site that published the robots.txt file. For more details on the structure of robots.txt files, see http://www.robotstxt.org/orig.html.

class robotparser.RobotFileParser(url='')

This class provides methods to read, parse and answer questions about the robots.txt file at url.

set_url(url)

Sets the URL referring to a robots.txt file.

read()

Reads the robots.txt URL and feeds it to the parser.

parse(lines)

Parses the lines argument.

can_fetch(useragent, url)

Returns True if the useragent is allowed to fetch the url according to the rules contained in the parsed robots.txt file.

mtime()

Returns the time the robots.txt file was last fetched. This is useful for long-running web spiders that need to check for new robots.txt files periodically.

modified()

Sets the time the robots.txt file was last fetched to the current time.

The following example demonstrates basic use of the RobotFileParser class.

```
>>> import robotparser
>>> rp = robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

13.4 netrc — netrc file processing

New in version 1.5.2.

Source code: Lib/netrc.py

The netrc class parses and encapsulates the netrc file format used by the Unix ftp program and other FTP clients.

class netrc.netrc([file])
: A netrc instance or subclass instance encapsulates data from a netrc file. The initialization argument, if present, specifies the file to parse. If no argument is given, the file .netrc in the user's home directory will be read. Parse errors will raise NetrcParseError with diagnostic information including the file name, line number, and terminating token. If no argument is specified on a POSIX system, the presence of passwords in the .netrc file will raise a NetrcParseError if the file ownership or permissions are insecure (owned by a user other than the user running the process, or accessible for read or write by any other user). This implements security behavior equivalent to that of ftp and other programs that use .netrc.

  Changed in version 2.7.6: Added the POSIX permissions check.

exception netrc.NetrcParseError
: Exception raised by the netrc class when syntactical errors are encountered in source text. Instances of this exception provide three interesting attributes: msg is a textual explanation of the error, filename is the name of the source file, and lineno gives the line number on which the error was found.

13.4.1 netrc Objects

A netrc instance has the following methods:

netrc.authenticators(host)
: Return a 3-tuple (login, account, password) of authenticators for host. If the netrc file did not contain an entry for the given host, return the tuple associated with the 'default' entry. If neither matching host nor default entry is available, return None.

netrc.__repr__()
: Dump the class data as a string in the format of a netrc file. (This discards comments and may reorder the entries.)

Instances of netrc have public instance variables:

netrc.hosts
: Dictionary mapping host names to (login, account, password) tuples. The 'default' entry, if any, is represented as a pseudo-host by that name.

netrc.macros
: Dictionary mapping macro names to string lists.

Note: Passwords are limited to a subset of the ASCII character set. Versions of this module prior to 2.3 were extremely limited. Starting with 2.3, all ASCII punctuation is allowed in passwords. However, note that whitespace and non-printable characters are not allowed in passwords. This is a limitation of the way the .netrc file is parsed and may be removed in the future.

13.5 xdrlib — Encode and decode XDR data

Source code: Lib/xdrlib.py

The xdrlib module supports the External Data Representation Standard as described in RFC 1014, written by Sun Microsystems, Inc. June 1987. It supports most of the data types described in the RFC.

The xdrlib module defines two classes, one for packing variables into XDR representation, and another for unpacking from XDR representation. There are also two exception classes.

class xdrlib.Packer
: Packer is the class for packing data into XDR representation. The Packer class is instantiated with no arguments.

class xdrlib.Unpacker(data)
: Unpacker is the complementary class which unpacks XDR data values from a string buffer. The input buffer is given as data.

See also:

RFC 1014 - XDR: External Data Representation Standard This RFC defined the encoding of data which was XDR at the time this module was originally written. It has apparently been obsoleted by RFC 1832.

RFC 1832 - XDR: External Data Representation Standard Newer RFC that provides a revised definition of XDR.

13.5.1 Packer Objects

Packer instances have the following methods:

Packer.get_buffer()
: Returns the current pack buffer as a string.

Packer.reset()
: Resets the pack buffer to the empty string.

In general, you can pack any of the most common XDR data types by calling the appropriate pack_type() method. Each method takes a single argument, the value to pack. The following simple data type packing methods are supported: pack_uint(), pack_int(), pack_enum(), pack_bool(), pack_uhyper(), and pack_hyper().

Packer.pack_float(value)
: Packs the single-precision floating point number value.

Packer.pack_double(value)
: Packs the double-precision floating point number value.

The following methods support packing strings, bytes, and opaque data:

Packer.pack_fstring(n, s)
: Packs a fixed length string, s. n is the length of the string but it is not packed into the data buffer. The string is padded with null bytes if necessary to guaranteed 4 byte alignment.

Packer.pack_fopaque(n, data)
: Packs a fixed length opaque data stream, similarly to pack_fstring().

Packer.pack_string(s)
: Packs a variable length string, s. The length of the string is first packed as an unsigned integer, then the string data is packed with pack_fstring().

Packer.pack_opaque(data)
: Packs a variable length opaque data string, similarly to pack_string().

Packer.pack_bytes(bytes)
: Packs a variable length byte stream, similarly to pack_string().

The following methods support packing arrays and lists:

Packer.pack_list(list, pack_item)
: Packs a list of homogeneous items. This method is useful for lists with an indeterminate size; i.e. the size is not available until the entire list has been walked. For each item in the list, an unsigned integer 1 is packed first, followed by the data value from the list. pack_item is the function that is called to pack the individual item. At the end of the list, an unsigned integer 0 is packed.

  For example, to pack a list of integers, the code might appear like this:

```
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

Packer.pack_farray(n, array, pack_item)
: Packs a fixed length list (array) of homogeneous items. n is the length of the list; it is not packed into the buffer, but a ValueError exception is raised if len(array) is not equal to n. As above, pack_item is the function used to pack each element.

Packer.pack_array(list, pack_item)
: Packs a variable length list of homogeneous items. First, the length of the list is packed as an unsigned integer, then each element is packed as in pack_farray() above.

13.5.2 Unpacker Objects

The Unpacker class offers the following methods:

Unpacker.reset(data)
: Resets the string buffer with the given data.

Unpacker.get_position()
: Returns the current unpack position in the data buffer.

Unpacker.set_position(position)
: Sets the data buffer unpack position to position. You should be careful about using get_position() and set_position().

Unpacker.get_buffer()
: Returns the current unpack data buffer as a string.

Unpacker.done()
: Indicates unpack completion. Raises an Error exception if all of the data has not been unpacked.

In addition, every data type that can be packed with a Packer, can be unpacked with an Unpacker. Unpacking methods are of the form unpack_type(), and take no arguments. They return the unpacked object.

Unpacker.unpack_float()
: Unpacks a single-precision floating point number.

Unpacker.unpack_double()
: Unpacks a double-precision floating point number, similarly to unpack_float().

In addition, the following methods unpack strings, bytes, and opaque data:

Unpacker.unpack_fstring(n)
: Unpacks and returns a fixed length string. n is the number of characters expected. Padding with null bytes to guaranteed 4 byte alignment is assumed.

Unpacker.unpack_fopaque(n)
: Unpacks and returns a fixed length opaque data stream, similarly to unpack_fstring().

Unpacker.unpack_string()
: Unpacks and returns a variable length string. The length of the string is first unpacked as an unsigned integer, then the string data is unpacked with unpack_fstring().

Unpacker.unpack_opaque()
: Unpacks and returns a variable length opaque data string, similarly to unpack_string().

Unpacker.unpack_bytes()
: Unpacks and returns a variable length byte stream, similarly to unpack_string().

The following methods support unpacking arrays and lists:

Unpacker.unpack_list(unpack_item)
: Unpacks and returns a list of homogeneous items. The list is unpacked one element at a time by first unpacking an unsigned integer flag. If the flag is 1, then the item is unpacked and appended to the list. A flag of 0 indicates the end of the list. unpack_item is the function that is called to unpack the items.

Unpacker.unpack_farray(n, unpack_item)
: Unpacks and returns (as a list) a fixed length array of homogeneous items. n is number of list elements to expect in the buffer. As above, unpack_item is the function used to unpack each element.

Unpacker.unpack_array(unpack_item)
: Unpacks and returns a variable length list of homogeneous items. First, the length of the list is unpacked as an unsigned integer, then each element is unpacked as in unpack_farray() above.

13.5.3 Exceptions

Exceptions in this module are coded as class instances:

exception xdrlib.Error
: The base exception class. Error has a single public attribute msg containing the description of the error.

exception xdrlib.ConversionError
: Class derived from Error. Contains no additional instance variables.

Here is an example of how you would catch one of these exceptions:

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print 'packing the double failed:', instance.msg
```

13.6 plistlib — Generate and parse Mac OS X .plist files

Changed in version 2.6: This module was previously only available in the Mac-specific library, it is now available for all platforms.

Source code: Lib/plistlib.py

This module provides an interface for reading and writing the "property list" XML files used mainly by Mac OS X.

The property list (.plist) file format is a simple XML pickle supporting basic object types, like dictionaries, lists, numbers and strings. Usually the top level object is a dictionary.

Values can be strings, integers, floats, booleans, tuples, lists, dictionaries (but only with string keys), Data or datetime.datetime objects. String values (including dictionary keys) may be unicode strings – they will be written out as UTF-8.

The <data> plist type is supported through the Data class. This is a thin wrapper around a Python string. Use Data if your strings contain control characters.

See also:

PList manual page Apple's documentation of the file format.

This module defines the following functions:

plistlib.readPlist(pathOrFile)
: Read a plist file. pathOrFile may either be a file name or a (readable) file object. Return the unpacked root object (which usually is a dictionary).

  The XML data is parsed using the Expat parser from xml.parsers.expat – see its documentation for possible exceptions on ill-formed XML. Unknown elements will simply be ignored by the plist parser.

plistlib.writePlist(rootObject, pathOrFile)
: Write rootObject to a plist file. pathOrFile may either be a file name or a (writable) file object.

  A TypeError will be raised if the object is of an unsupported type or a container that contains objects of unsupported types.

plistlib.readPlistFromString(data)
: Read a plist from a string. Return the root object.

plistlib.writePlistToString(rootObject)
: Return rootObject as a plist-formatted string.

plistlib.readPlistFromResource(path, restype='plst', resid=0)
: Read a plist from the resource with type restype from the resource fork of path. Availability: Mac OS X.

  Note: In Python 3.x, this function has been removed.

  ---

plistlib.writePlistToResource(rootObject, path, restype='plst', resid=0)
: Write rootObject as a resource with type restype to the resource fork of path. Availability: Mac OS X.

  Note: In Python 3.x, this function has been removed.

The following class is available:

class plistlib.Data(data)
: Return a "data" wrapper object around the string data. This is used in functions converting from/to plists to represent the <data> type available in plists.

  It has one attribute, data, that can be used to retrieve the Python string stored in it.

13.6.1 Examples

Generating a plist:

```
pl = dict(
    aString="Doodah",
    aList=["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict=dict(
        anotherString="<hello & hi there!>",
        aUnicodeValue=u'M\xe4ssig, Ma\xdf',
        aTrueValue=True,
        aFalseValue=False,
    ),
    someData = Data("<binary gunk>"),
    someMoreData = Data("<lots of binary gunk>" * 10),
    aDate = datetime.datetime.fromtimestamp(time.mktime(time.gmtime())),
)
# unicode keys are possible, but a little awkward to use:
pl[u'\xc5benraa'] = "That was a unicode key."
writePlist(pl, fileName)
```

Parsing a plist:

```
pl = readPlist(pathOrFile)
print pl["aKey"]
```

CHAPTER

FOURTEEN

CRYPTOGRAPHIC SERVICES

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. Here's an overview:

14.1 hashlib — Secure hashes and message digests

New in version 2.5.

Source code: Lib/hashlib.py

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 (defined in FIPS 180-2) as well as RSA's MD5 algorithm (defined in Internet RFC 1321). The terms secure hash and message digest are interchangeable. Older algorithms were called message digests. The modern term is secure hash.

Note: If you want the adler32 or crc32 hash functions, they are available in the zlib module.

Warning: Some algorithms have known hash collision weaknesses, refer to the "See also" section at the end.

There is one constructor method named for each type of hash. All return a hash object with the same simple interface. For example: use sha1() to create a SHA1 hash object. You can now feed this object with arbitrary strings using the update() method. At any point you can ask it for the digest of the concatenation of the strings fed to it so far using the digest() or hexdigest() methods.

Constructors for hash algorithms that are always present in this module are md5(), sha1(), sha224(), sha256(), sha384(), and sha512(). Additional algorithms may also be available depending upon the OpenSSL library that Python uses on your platform.

For example, to obtain the digest of the string `'Nobody inspects the spammish repetition'`:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

(continues on next page)

(continued from previous page)

```
>>> m.digest_size
16
>>> m.block_size
64
```

More condensed:

```
>>> hashlib.sha224("Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

A generic new() constructor that takes the string name of the desired algorithm as its first parameter also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer. The named constructors are much faster than new() and should be preferred.

Using new() with an algorithm provided by OpenSSL:

```
>>> h = hashlib.new('ripemd160')
>>> h.update("Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

This module provides the following constant attribute:

hashlib.algorithms
: A tuple providing the names of the hash algorithms guaranteed to be supported by this module.

  New in version 2.7.

hashlib.algorithms_guaranteed
: A set containing the names of the hash algorithms guaranteed to be supported by this module on all platforms.

  New in version 2.7.9.

hashlib.algorithms_available
: A set containing the names of the hash algorithms that are available in the running Python interpreter. These names will be recognized when passed to new(). algorithms_guaranteed will always be a subset. The same algorithm may appear multiple times in this set under different names (thanks to OpenSSL).

  New in version 2.7.9.

The following values are provided as constant attributes of the hash objects returned by the constructors:

hash.digest_size
: The size of the resulting hash in bytes.

hash.block_size
: The internal block size of the hash algorithm in bytes.

A hash object has the following methods:

hash.update(arg)
: Update the hash object with the string arg. Repeated calls are equivalent to a single call with the concatenation of all the arguments: m.update(a); m.update(b) is equivalent to m.update(a+b).

  Changed in version 2.7: The Python GIL is released to allow other threads to run while hash updates on data larger than 2048 bytes is taking place when using hash algorithms supplied by OpenSSL.

hash.digest()
: Return the digest of the strings passed to the update() method so far. This is a string of digest_size bytes which may contain non-ASCII characters, including null bytes.

hash.hexdigest()
: Like digest() except the digest is returned as a string of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

hash.copy()
: Return a copy ("clone") of the hash object. This can be used to efficiently compute the digests of strings that share a common initial substring.

14.1.1 Key derivation

Key derivation and key stretching algorithms are designed for secure password hashing. Naive algorithms such as sha1(password) are not resistant against brute-force attacks. A good password hashing function must be tunable, slow, and include a salt.

hashlib.pbkdf2_hmac(name, password, salt, rounds, dklen=None)
: The function provides PKCS#5 password-based key derivation function 2. It uses HMAC as pseudo-random function.

  The string name is the desired name of the hash digest algorithm for HMAC, e.g. 'sha1' or 'sha256'. password and salt are interpreted as buffers of bytes. Applications and libraries should limit password to a sensible value (e.g. 1024). salt should be about 16 or more bytes from a proper source, e.g. os.urandom().

  The number of rounds should be chosen based on the hash algorithm and computing power. As of 2013, at least 100,000 rounds of SHA-256 is suggested.

  dklen is the length of the derived key. If dklen is None then the digest size of the hash algorithm name is used, e.g. 64 for SHA-512.

```
>>> import hashlib, binascii
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
>>> binascii.hexlify(dk)
b'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

  New in version 2.7.8.

Note: A fast implementation of pbkdf2_hmac is available with OpenSSL. The Python implementation uses an inline version of hmac. It is about three times slower and doesn't release the GIL.

See also:

Module hmac A module to generate message authentication codes using hashes.

Module base64 Another way to encode binary hashes for non-binary environments.

http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf The FIPS 180-2 publication on Secure Hash Algorithms.

https://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms
: Wikipedia article with information on which algorithms have known issues and what that means regarding their use.

14.2 hmac — Keyed-Hashing for Message Authentication

New in version 2.2.

Source code: Lib/hmac.py

This module implements the HMAC algorithm as described by RFC 2104.

hmac.new(key[, msg[, digestmod]])
: Return a new hmac object. If msg is present, the method call update(msg) is made. digestmod is the digest constructor or module for the HMAC object to use. It defaults to the hashlib.md5 constructor.

An HMAC object has the following methods:

HMAC.update(msg)
: Update the hmac object with the string msg. Repeated calls are equivalent to a single call with the concatenation of all the arguments: m.update(a); m.update(b) is equivalent to m.update(a + b).

HMAC.digest()
: Return the digest of the strings passed to the update() method so far. This string will be the same length as the digest_size of the digest given to the constructor. It may contain non-ASCII characters, including NUL bytes.

> Warning: When comparing the output of digest() to an externally-supplied digest during a verification routine, it is recommended to use the compare_digest() function instead of the == operator to reduce the vulnerability to timing attacks.

HMAC.hexdigest()
: Like digest() except the digest is returned as a string twice the length containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

> Warning: When comparing the output of hexdigest() to an externally-supplied digest during a verification routine, it is recommended to use the compare_digest() function instead of the == operator to reduce the vulnerability to timing attacks.

HMAC.copy()
: Return a copy ("clone") of the hmac object. This can be used to efficiently compute the digests of strings that share a common initial substring.

This module also provides the following helper function:

hmac.compare_digest(a, b)
: Return a == b. This function uses an approach designed to prevent timing analysis by avoiding content-based short circuiting behaviour, making it appropriate for cryptography. a and b must both be of the same type: either unicode or a bytes-like object.

  Note: If a and b are of different lengths, or if an error occurs, a timing attack could theoretically reveal information about the types and lengths of a and b—but not their values.

  New in version 2.7.7.

See also:

Module hashlib The Python module providing secure hash functions.

14.3 md5 — MD5 message digest algorithm

Deprecated since version 2.5: Use the hashlib module instead.

This module implements the interface to RSA's MD5 message digest algorithm (see also Internet RFC 1321). Its use is quite straightforward: use new() to create an md5 object. You can now feed this object with arbitrary strings using the update() method, and at any point you can ask it for the digest (a strong kind of 128-bit checksum, a.k.a. "fingerprint") of the concatenation of the strings fed to it so far using the digest() method.

For example, to obtain the digest of the string `'Nobody inspects the spammish repetition'`:

```
>>> import md5
>>> m = md5.new()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

More condensed:

```
>>> md5.new("Nobody inspects the spammish repetition").digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

The following values are provided as constants in the module and as attributes of the md5 objects returned by new():

md5.digest_size
: The size of the resulting digest in bytes. This is always 16.

The md5 module provides the following functions:

md5.new([arg])
: Return a new md5 object. If arg is present, the method call update(arg) is made.

md5.md5([arg])
: For backward compatibility reasons, this is an alternative name for the new() function.

An md5 object has the following methods:

md5.update(arg)
: Update the md5 object with the string arg. Repeated calls are equivalent to a single call with the concatenation of all the arguments: m.update(a); m.update(b) is equivalent to m.update(a+b).

md5.digest()
: Return the digest of the strings passed to the update() method so far. This is a 16-byte string which may contain non-ASCII characters, including null bytes.

md5.hexdigest()
: Like digest() except the digest is returned as a string of length 32, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

md5.copy()
: Return a copy ("clone") of the md5 object. This can be used to efficiently compute the digests of strings that share a common initial substring.

See also:

Module sha Similar module implementing the Secure Hash Algorithm (SHA). The SHA algorithm is considered a more secure hash.

14.4 sha — SHA-1 message digest algorithm

Deprecated since version 2.5: Use the hashlib module instead.

This module implements the interface to NIST's secure hash algorithm, known as SHA-1. SHA-1 is an improved version of the original SHA hash algorithm. It is used in the same way as the md5 module: use new() to create an sha object, then feed this object with arbitrary strings using the update() method, and at any point you can ask it for the digest of the concatenation of the strings fed to it so far. SHA-1 digests are 160 bits instead of MD5's 128 bits.

sha.new([string])
: Return a new sha object. If string is present, the method call update(string) is made.

The following values are provided as constants in the module and as attributes of the sha objects returned by new():

sha.blocksize
: Size of the blocks fed into the hash function; this is always 1. This size is used to allow an arbitrary string to be hashed.

sha.digest_size
: The size of the resulting digest in bytes. This is always 20.

An sha object has the same methods as md5 objects:

sha.update(arg)
: Update the sha object with the string arg. Repeated calls are equivalent to a single call with the concatenation of all the arguments: m.update(a); m.update(b) is equivalent to m.update(a+b).

sha.digest()
: Return the digest of the strings passed to the update() method so far. This is a 20-byte string which may contain non-ASCII characters, including null bytes.

sha.hexdigest()
: Like digest() except the digest is returned as a string of length 40, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

sha.copy()
: Return a copy ("clone") of the sha object. This can be used to efficiently compute the digests of strings that share a common initial substring.

See also:

Secure Hash Standard The Secure Hash Algorithm is defined by NIST document FIPS PUB 180-2: Secure Hash Standard, published in August 2002.

Cryptographic Toolkit (Secure Hashing) Links from NIST to various information on secure hashing.

CHAPTER

FIFTEEN

GENERIC OPERATING SYSTEM SERVICES

The modules described in this chapter provide interfaces to operating system features that are available on (almost) all operating systems, such as files and a clock. The interfaces are generally modeled after the Unix or C interfaces, but they are available on most other systems as well. Here's an overview:

15.1 os — Miscellaneous operating system interfaces

This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file see open(), if you want to manipulate paths, see the os.path module, and if you want to read all the lines in all the files on the command line see the fileinput module. For creating temporary files and directories see the tempfile module, and for high-level file and directory handling see the shutil module.

Notes on the availability of these functions:

- The design of all built-in operating system dependent modules of Python is such that as long as the same functionality is available, it uses the same interface; for example, the function os.stat(path) returns stat information about path in the same format (which happens to have originated with the POSIX interface).
- Extensions peculiar to a particular operating system are also available through the os module, but using them is of course a threat to portability.
- An "Availability: Unix" note means that this function is commonly found on Unix systems. It does not make any claims about its existence on a specific operating system.
- If not separately noted, all functions that claim "Availability: Unix" are supported on Mac OS X, which builds on a Unix core.

Note: All functions in this module raise OSError in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.

---

exception os.error
: An alias for the built-in OSError exception.

os.name
: The name of the operating system dependent module imported. The following names have currently been registered: `'posix'`, `'nt'`, `'os2'`, `'ce'`, `'java'`, `'riscos'`.

  See also:

  sys.platform has a finer granularity. os.uname() gives system-dependent version information.

  The platform module provides detailed checks for the system's identity.

15.1.1 Process Parameters

These functions and data items provide information and operate on the current process and user.

os.environ
: A mapping object representing the string environment. For example, environ['HOME'] is the pathname of your home directory (on some platforms), and is equivalent to getenv("HOME") in C.

  This mapping is captured the first time the os module is imported, typically during Python startup as part of processing site.py. Changes to the environment made after this time are not reflected in os.environ, except for changes made by modifying os.environ directly.

  If the platform supports the putenv() function, this mapping may be used to modify the environment as well as query the environment. putenv() will be called automatically when the mapping is modified.

  Note: Calling putenv() directly does not change os.environ, so it's better to modify os.environ.

  Note: On some platforms, including FreeBSD and Mac OS X, setting environ may cause memory leaks. Refer to the system documentation for putenv().

  If putenv() is not provided, a modified copy of this mapping may be passed to the appropriate process-creation functions to cause child processes to use a modified environment.

  If the platform supports the unsetenv() function, you can delete items in this mapping to unset environment variables. unsetenv() will be called automatically when an item is deleted from os.environ, and when one of the pop() or clear() methods is called.

  Changed in version 2.6: Also unset environment variables when calling os.environ.clear() and os.environ.pop().

os.chdir(path)
os.fchdir(fd)
os.getcwd()
: These functions are described in Files and Directories.

os.ctermid()
: Return the filename corresponding to the controlling terminal of the process.

  Availability: Unix.

os.getegid()
: Return the effective group id of the current process. This corresponds to the "set id" bit on the file being executed in the current process.

  Availability: Unix.

os.geteuid()
: Return the current process's effective user id.

  Availability: Unix.

os.getgid()
: Return the real group id of the current process.

  Availability: Unix.

os.getgroups()
: Return list of supplemental group ids associated with the current process.

Availability: Unix.

Note: On Mac OS X, getgroups() behavior differs somewhat from other Unix platforms. If the Python interpreter was built with a deployment target of 10.5 or earlier, getgroups() returns the list of effective group ids associated with the current user process; this list is limited to a system-defined number of entries, typically 16, and may be modified by calls to setgroups() if suitably privileged. If built with a deployment target greater than 10.5, getgroups() returns the current group access list for the user associated with the effective user id of the process; the group access list may change over the lifetime of the process, it is not affected by calls to setgroups(), and its length is not limited to 16. The deployment target value, MACOSX_DEPLOYMENT_TARGET, can be obtained with sysconfig.get_config_var().

---

os.initgroups(username, gid)
Call the system initgroups() to initialize the group access list with all of the groups of which the specified username is a member, plus the specified group id.

Availability: Unix.

New in version 2.7.

os.getlogin()
Return the name of the user logged in on the controlling terminal of the process. For most purposes, it is more useful to use the environment variable LOGNAME to find out who the user is, or pwd.getpwuid(os.getuid())[0] to get the login name of the process's real user id.

Availability: Unix.

os.getpgid(pid)
Return the process group id of the process with process id pid. If pid is 0, the process group id of the current process is returned.

Availability: Unix.

New in version 2.3.

os.getpgrp()
Return the id of the current process group.

Availability: Unix.

os.getpid()
Return the current process id.

Availability: Unix, Windows.

os.getppid()
Return the parent's process id.

Availability: Unix.

os.getresuid()
Return a tuple (ruid, euid, suid) denoting the current process's real, effective, and saved user ids.

Availability: Unix.

New in version 2.7.

os.getresgid()
Return a tuple (rgid, egid, sgid) denoting the current process's real, effective, and saved group ids.

Availability: Unix.

New in version 2.7.

os.getuid()
Return the current process's real user id.

Availability: Unix.

os.getenv(varname[, value])
Return the value of the environment variable varname if it exists, or value if it doesn't. value defaults to None.

Availability: most flavors of Unix, Windows.

os.putenv(varname, value)
Set the environment variable named varname to the string value. Such changes to the environment affect subprocesses started with os.system(), popen() or fork() and execv().

Availability: most flavors of Unix, Windows.

Note: On some platforms, including FreeBSD and Mac OS X, setting environ may cause memory leaks. Refer to the system documentation for putenv.

When putenv() is supported, assignments to items in os.environ are automatically translated into corresponding calls to putenv(); however, calls to putenv() don't update os.environ, so it is actually preferable to assign to items of os.environ.

os.setegid(egid)
Set the current process's effective group id.

Availability: Unix.

os.seteuid(euid)
Set the current process's effective user id.

Availability: Unix.

os.setgid(gid)
Set the current process' group id.

Availability: Unix.

os.setgroups(groups)
Set the list of supplemental group ids associated with the current process to groups. groups must be a sequence, and each element must be an integer identifying a group. This operation is typically available only to the superuser.

Availability: Unix.

New in version 2.2.

Note: On Mac OS X, the length of groups may not exceed the system-defined maximum number of effective group ids, typically 16. See the documentation for getgroups() for cases where it may not return the same group list set by calling setgroups().

---

os.setpgrp()
Call the system call setpgrp() or setpgrp(0, 0) depending on which version is implemented (if any). See the Unix manual for the semantics.

Availability: Unix.

os.setpgid(pid, pgrp)
Call the system call setpgid() to set the process group id of the process with id pid to the process group with id pgrp. See the Unix manual for the semantics.

Availability: Unix.

os.setregid(rgid, egid)
Set the current process's real and effective group ids.

Availability: Unix.

os.setresgid(rgid, egid, sgid)
Set the current process's real, effective, and saved group ids.

Availability: Unix.

New in version 2.7.

os.setresuid(ruid, euid, suid)
Set the current process's real, effective, and saved user ids.

Availability: Unix.

New in version 2.7.

os.setreuid(ruid, euid)
Set the current process's real and effective user ids.

Availability: Unix.

os.getsid(pid)
Call the system call getsid(). See the Unix manual for the semantics.

Availability: Unix.

New in version 2.4.

os.setsid()
Call the system call setsid(). See the Unix manual for the semantics.

Availability: Unix.

os.setuid(uid)
Set the current process's user id.

Availability: Unix.

os.strerror(code)
Return the error message corresponding to the error code in code. On platforms where strerror() returns NULL when given an unknown error number, ValueError is raised.

Availability: Unix, Windows.

os.umask(mask)
Set the current numeric umask and return the previous umask.

Availability: Unix, Windows.

os.uname()
Return a 5-tuple containing information identifying the current operating system. The tuple contains 5 strings: (sysname, nodename, release, version, machine). Some systems truncate the nodename to 8 characters or to the leading component; a better way to get the hostname is socket.gethostname() or even socket.gethostbyaddr(socket.gethostname()).

Availability: recent flavors of Unix.

os.unsetenv(varname)
Unset (delete) the environment variable named varname. Such changes to the environment affect subprocesses started with os.system(), popen() or fork() and execv().

When unsetenv() is supported, deletion of items in os.environ is automatically translated into a corresponding call to unsetenv(); however, calls to unsetenv() don't update os.environ, so it is actually preferable to delete items of os.environ.

Availability: most flavors of Unix, Windows.

15.1.2 File Object Creation

These functions create new file objects. (See also open().)

os.fdopen(fd[, mode[, bufsize]])
Return an open file object connected to the file descriptor fd. The mode and bufsize arguments have the same meaning as the corresponding arguments to the built-in open() function. If fdopen() raises an exception, it leaves fd untouched (unclosed).

Availability: Unix, Windows.

Changed in version 2.3: When specified, the mode argument must now start with one of the letters 'r', 'w', or 'a', otherwise a ValueError is raised.

Changed in version 2.5: On Unix, when the mode argument starts with 'a', the O_APPEND flag is set on the file descriptor (which the fdopen() implementation already does on most platforms).

os.popen(command[, mode[, bufsize]])
Open a pipe to or from command. The return value is an open file object connected to the pipe, which can be read or written depending on whether mode is 'r' (default) or 'w'. The bufsize argument has the same meaning as the corresponding argument to the built-in open() function. The exit status of the command (encoded in the format specified for wait()) is available as the return value of the close() method of the file object, except that when the exit status is zero (termination without errors), None is returned.

Availability: Unix, Windows.

Deprecated since version 2.6: This function is obsolete. Use the subprocess module. Check especially the Replacing Older Functions with the subprocess Module section.

Changed in version 2.0: This function worked unreliably under Windows in earlier versions of Python. This was due to the use of the _popen() function from the libraries provided with Windows. Newer versions of Python do not use the broken implementation from the Windows libraries.

os.tmpfile()
Return a new file object opened in update mode (w+b). The file has no directory entries associated with it and will be automatically deleted once there are no file descriptors for the file.

Availability: Unix, Windows.

There are a number of different popen*() functions that provide slightly different ways to create subprocesses.

Deprecated since version 2.6: All of the popen*() functions are obsolete. Use the subprocess module.

For each of the popen*() variants, if bufsize is specified, it specifies the buffer size for the I/O pipes. mode, if provided, should be the string 'b' or 't'; on Windows this is needed to determine whether the file objects should be opened in binary or text mode. The default value for mode is 't'.

Also, for each of these variants, on Unix, cmd may be a sequence, in which case arguments will be passed directly to the program without shell intervention (as with os.spawnv()). If cmd is a string it will be passed to the shell (as with os.system()).

These methods do not make it possible to retrieve the exit status from the child processes. The only way to control the input and output streams and also retrieve the return codes is to use the subprocess module; these are only available on Unix.

For a discussion of possible deadlock conditions related to the use of these functions, see Flow Control Issues.

os.popen2(cmd[, mode[, bufsize]])
Execute cmd as a sub-process and return the file objects (child_stdin, child_stdout).

Deprecated since version 2.6: This function is obsolete. Use the subprocess module. Check especially the Replacing Older Functions with the subprocess Module section.

Availability: Unix, Windows.

New in version 2.0.

os.popen3(cmd[, mode[, bufsize]])
Execute cmd as a sub-process and return the file objects (child_stdin, child_stdout, child_stderr).

Deprecated since version 2.6: This function is obsolete. Use the subprocess module. Check especially the Replacing Older Functions with the subprocess Module section.

Availability: Unix, Windows.

New in version 2.0.

os.popen4(cmd[, mode[, bufsize]])
Execute cmd as a sub-process and return the file objects (child_stdin, child_stdout_and_stderr).

Deprecated since version 2.6: This function is obsolete. Use the subprocess module. Check especially the Replacing Older Functions with the subprocess Module section.

Availability: Unix, Windows.

New in version 2.0.

(Note that child_stdin, child_stdout, and child_stderr are named from the point of view of the child process, so child_stdin is the child's standard input.)

This functionality is also available in the popen2 module using functions of the same names, but the return values of those functions have a different order.

15.1.3 File Descriptor Operations

These functions operate on I/O streams referenced using file descriptors.

File descriptors are small integers corresponding to a file that has been opened by the current process. For example, standard input is usually file descriptor 0, standard output is 1, and standard error is 2. Further files opened by a process will then be assigned 3, 4, 5, and so forth. The name "file descriptor" is slightly deceptive; on Unix platforms, sockets and pipes are also referenced by file descriptors.

The fileno() method can be used to obtain the file descriptor associated with a file object when required. Note that using the file descriptor directly will bypass the file object methods, ignoring aspects such as internal buffering of data.

os.close(fd)
Close file descriptor fd.

Availability: Unix, Windows.

Note: This function is intended for low-level I/O and must be applied to a file descriptor as returned by os.open() or pipe(). To close a "file object" returned by the built-in function open() or by popen() or fdopen(), use its close() method.

os.closerange(fd_low, fd_high)
Close all file descriptors from fd_low (inclusive) to fd_high (exclusive), ignoring errors. Equivalent to:

```
for fd in xrange(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

Availability: Unix, Windows.

New in version 2.6.

os.dup(fd)
Return a duplicate of file descriptor fd.

Availability: Unix, Windows.

os.dup2(fd, fd2)
Duplicate file descriptor fd to fd2, closing the latter first if necessary.

Availability: Unix, Windows.

os.fchmod(fd, mode)
Change the mode of the file given by fd to the numeric mode. See the docs for chmod() for possible values of mode.

Availability: Unix.

New in version 2.6.

os.fchown(fd, uid, gid)
Change the owner and group id of the file given by fd to the numeric uid and gid. To leave one of the ids unchanged, set it to -1.

Availability: Unix.

New in version 2.6.

os.fdatasync(fd)
Force write of file with filedescriptor fd to disk. Does not force update of metadata.

Availability: Unix.

Note: This function is not available on MacOS.

os.fpathconf(fd, name)
Return system configuration information relevant to an open file. name specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the pathconf_names dictionary. For configuration variables not included in that mapping, passing an integer for name is also accepted.

If name is a string and is not known, ValueError is raised. If a specific value for name is not supported by the host system, even if it is included in pathconf_names, an OSError is raised with errno.EINVAL for the error number.

Availability: Unix.

os.fstat(fd)
Return status for file descriptor fd, like stat().

Availability: Unix, Windows.

os.fstatvfs(fd)
Return information about the filesystem containing the file associated with file descriptor fd, like statvfs().

Availability: Unix.

os.fsync(fd)
Force write of file with filedescriptor fd to disk. On Unix, this calls the native fsync() function; on Windows, the MS _commit() function.

If you're starting with a Python file object f, first do f.flush(), and then do os.fsync(f.fileno()), to ensure that all internal buffers associated with f are written to disk.

Availability: Unix, and Windows starting in 2.2.3.

os.ftruncate(fd, length)
Truncate the file corresponding to file descriptor fd, so that it is at most length bytes in size.

Availability: Unix.

os.isatty(fd)
Return True if the file descriptor fd is open and connected to a tty(-like) device, else False.

os.lseek(fd, pos, how)
Set the current position of file descriptor fd to position pos, modified by how: SEEK_SET or 0 to set the position relative to the beginning of the file; SEEK_CUR or 1 to set it relative to the current position; SEEK_END or 2 to set it relative to the end of the file. Return the new cursor position in bytes, starting from the beginning.

Availability: Unix, Windows.

os.SEEK_SET
os.SEEK_CUR
os.SEEK_END
Parameters to the lseek() function. Their values are 0, 1, and 2, respectively.

Availability: Windows, Unix.

New in version 2.5.

os.open(file, flags[, mode])
Open the file file and set various flags according to flags and possibly its mode according to mode. The default mode is 0777 (octal), and the current umask value is first masked out. Return the file descriptor for the newly opened file.

For a description of the flag and mode values, see the C run-time documentation; flag constants (like O_RDONLY and O_WRONLY) are defined in this module too (see open() flag constants). In particular, on Windows adding O_BINARY is needed to open files in binary mode.

Availability: Unix, Windows.

Note: This function is intended for low-level I/O. For normal usage, use the built-in function open(), which returns a "file object" with read() and write() methods (and many more). To wrap a file descriptor in a "file object", use fdopen().

os.openpty()
Open a new pseudo-terminal pair. Return a pair of file descriptors (master, slave) for the pty and the tty, respectively. For a (slightly) more portable approach, use the pty module.

Availability: some flavors of Unix.

os.pipe()
Create a pipe. Return a pair of file descriptors (r, w) usable for reading and writing, respectively.

Availability: Unix, Windows.

os.read(fd, n)
Read at most n bytes from file descriptor fd. Return a string containing the bytes read. If the end of the file referred to by fd has been reached, an empty string is returned.

Availability: Unix, Windows.

Note: This function is intended for low-level I/O and must be applied to a file descriptor as returned by os.open() or pipe(). To read a "file object" returned by the built-in function open() or by popen() or fdopen(), or sys.stdin, use its read() or readline() methods.

os.tcgetpgrp(fd)
Return the process group associated with the terminal given by fd (an open file descriptor as returned by os.open()).

Availability: Unix.

os.tcsetpgrp(fd, pg)
Set the process group associated with the terminal given by fd (an open file descriptor as returned by os.open()) to pg.

Availability: Unix.

os.ttyname(fd)
Return a string which specifies the terminal device associated with file descriptor fd. If fd is not associated with a terminal device, an exception is raised.

Availability: Unix.

os.write(fd, str)
Write the string str to file descriptor fd. Return the number of bytes actually written.

Availability: Unix, Windows.

Note: This function is intended for low-level I/O and must be applied to a file descriptor as returned by os.open() or pipe(). To write a "file object" returned by the built-in function open() or by popen() or fdopen(), or sys.stdout or sys.stderr, use its write() method.

open() flag constants

The following constants are options for the flags parameter to the open() function. They can be combined using the bitwise OR operator |. Some of them are not available on all platforms. For descriptions of their availability and use, consult the open(2) manual page on Unix or the MSDN on Windows.

os.O_RDONLY
os.O_WRONLY
os.O_RDWR
os.O_APPEND
os.O_CREAT
os.O_EXCL
os.O_TRUNC
The above constants are available on Unix and Windows.

os.O_DSYNC
os.O_RSYNC
os.O_SYNC
os.O_NDELAY
os.O_NONBLOCK
os.O_NOCTTY
The above constants are only available on Unix.

os.O_BINARY
os.O_NOINHERIT
os.O_SHORT_LIVED
os.O_TEMPORARY
os.O_RANDOM
os.O_SEQUENTIAL
os.O_TEXT
The above constants are only available on Windows.

os.O_ASYNC
os.O_DIRECT
os.O_DIRECTORY
os.O_NOFOLLOW
os.O_NOATIME
os.O_SHLOCK
os.O_EXLOCK
The above constants are extensions and not present if they are not defined by the C library.

15.1.4 Files and Directories

os.access(path, mode)
Use the real uid/gid to test for access to path. Note that most operations will use the effective uid/gid, therefore this routine can be used in a suid/sgid environment to test if the invoking user has the specified access to path. mode should be F_OK to test the existence of path, or it can be the inclusive OR of one or more of R_OK, W_OK, and X_OK to test permissions. Return True if access is allowed, False if not. See the Unix man page access(2) for more information.

Availability: Unix, Windows.

Note: Using access() to check if a user is authorized to e.g. open a file before actually doing so using open() creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it. It's preferable to use EAFP techniques. For example:

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

is better written as:

```
try:
    fp = open("myfile")
except IOError as e:
    if e.errno == errno.EACCES:
        return "some default data"
    # Not a permission error.
    raise
else:
    with fp:
        return fp.read()
```

Note: I/O operations may fail even when access() indicates that they would succeed, particularly for operations on network filesystems which may have permissions semantics beyond the usual POSIX permission-bit model.

---

os.F_OK
: Value to pass as the mode parameter of access() to test the existence of path.

os.R_OK
: Value to include in the mode parameter of access() to test the readability of path.

os.W_OK
: Value to include in the mode parameter of access() to test the writability of path.

os.X_OK
: Value to include in the mode parameter of access() to determine if path can be executed.

os.chdir(path)
: Change the current working directory to path.

  Availability: Unix, Windows.

os.fchdir(fd)
: Change the current working directory to the directory represented by the file descriptor fd. The descriptor must refer to an opened directory, not an open file.

  Availability: Unix.

  New in version 2.3.

os.getcwd()
: Return a string representing the current working directory.

  Availability: Unix, Windows.

os.getcwdu()
: Return a Unicode object representing the current working directory.

  Availability: Unix, Windows.

  New in version 2.3.

os.chflags(path, flags)
Set the flags of path to the numeric flags. flags may take a combination (bitwise OR) of the following values (as defined in the stat module):

- stat.UF_NODUMP
- stat.UF_IMMUTABLE
- stat.UF_APPEND
- stat.UF_OPAQUE
- stat.UF_NOUNLINK
- stat.UF_COMPRESSED
- stat.UF_HIDDEN
- stat.SF_ARCHIVED
- stat.SF_IMMUTABLE
- stat.SF_APPEND
- stat.SF_NOUNLINK
- stat.SF_SNAPSHOT

Availability: Unix.

New in version 2.6.

os.chroot(path)
Change the root directory of the current process to path. Availability: Unix.

New in version 2.2.

os.chmod(path, mode)
Change the mode of path to the numeric mode. mode may take one of the following values (as defined in the stat module) or bitwise ORed combinations of them:

- stat.S_ISUID
- stat.S_ISGID
- stat.S_ENFMT
- stat.S_ISVTX
- stat.S_IREAD
- stat.S_IWRITE
- stat.S_IEXEC
- stat.S_IRWXU
- stat.S_IRUSR
- stat.S_IWUSR
- stat.S_IXUSR
- stat.S_IRWXG
- stat.S_IRGRP
- stat.S_IWGRP
- stat.S_IXGRP

- stat.S_IRWXO
- stat.S_IROTH
- stat.S_IWOTH
- stat.S_IXOTH

Availability: Unix, Windows.

Note: Although Windows supports chmod(), you can only set the file's read-only flag with it (via the stat.S_IWRITE and stat.S_IREAD constants or a corresponding integer value). All other bits are ignored.

---

os.chown(path, uid, gid)
Change the owner and group id of path to the numeric uid and gid. To leave one of the ids unchanged, set it to -1.

Availability: Unix.

os.lchflags(path, flags)
Set the flags of path to the numeric flags, like chflags(), but do not follow symbolic links.

Availability: Unix.

New in version 2.6.

os.lchmod(path, mode)
Change the mode of path to the numeric mode. If path is a symlink, this affects the symlink rather than the target. See the docs for chmod() for possible values of mode.

Availability: Unix.

New in version 2.6.

os.lchown(path, uid, gid)
Change the owner and group id of path to the numeric uid and gid. This function will not follow symbolic links.

Availability: Unix.

New in version 2.3.

os.link(source, link_name)
Create a hard link pointing to source named link_name.

Availability: Unix.

os.listdir(path)
Return a list containing the names of the entries in the directory given by path. The list is in arbitrary order. It does not include the special entries `'.'` and `'..'` even if they are present in the directory.

Availability: Unix, Windows.

Changed in version 2.3: On Windows NT/2k/XP and Unix, if path is a Unicode object, the result will be a list of Unicode objects. Undecodable filenames will still be returned as string objects.

os.lstat(path)
Perform the equivalent of an lstat() system call on the given path. Similar to stat(), but does not follow symbolic links. On platforms that do not support symbolic links, this is an alias for stat().

os.mkfifo(path[, mode])
Create a FIFO (a named pipe) named path with numeric mode mode. The default mode is 0666 (octal). The current umask value is first masked out from the mode.

Availability: Unix.

FIFOs are pipes that can be accessed like regular files. FIFOs exist until they are deleted (for example with os.unlink()). Generally, FIFOs are used as rendezvous between "client" and "server" type processes: the server opens the FIFO for reading, and the client opens it for writing. Note that mkfifo() doesn't open the FIFO — it just creates the rendezvous point.

os.mknod(filename[, mode=0600[, device=0]])
Create a filesystem node (file, device special file or named pipe) named filename. mode specifies both the permissions to use and the type of node to be created, being combined (bitwise OR) with one of stat.S_IFREG, stat.S_IFCHR, stat.S_IFBLK, and stat.S_IFIFO (those constants are available in stat). For stat.S_IFCHR and stat.S_IFBLK, device defines the newly created device special file (probably using os.makedev()), otherwise it is ignored.

New in version 2.3.

os.major(device)
Extract the device major number from a raw device number (usually the st_dev or st_rdev field from stat).

New in version 2.3.

os.minor(device)
Extract the device minor number from a raw device number (usually the st_dev or st_rdev field from stat).

New in version 2.3.

os.makedev(major, minor)
Compose a raw device number from the major and minor device numbers.

New in version 2.3.

os.mkdir(path[, mode])
Create a directory named path with numeric mode mode. The default mode is 0777 (octal). If the directory already exists, OSError is raised.

On some systems, mode is ignored. Where it is used, the current umask value is first masked out. If bits other than the last 9 (i.e. the last 3 digits of the octal representation of the mode) are set, their meaning is platform-dependent. On some platforms, they are ignored and you should call chmod() explicitly to set them.

It is also possible to create temporary directories; see the tempfile module's tempfile.mkdtemp() function.

Availability: Unix, Windows.

os.makedirs(path[, mode])
Recursive directory creation function. Like mkdir(), but makes all intermediate-level directories needed to contain the leaf directory. Raises an error exception if the leaf directory already exists or cannot be created. The default mode is 0777 (octal).

The mode parameter is passed to mkdir(); see the mkdir() description for how it is interpreted.

Note: makedirs() will become confused if the path elements to create include os.pardir.

New in version 1.5.2.

Changed in version 2.3: This function now handles UNC paths correctly.

os.pathconf(path, name)
Return system configuration information relevant to a named file. name specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the pathconf_names dictionary. For configuration variables not included in that mapping, passing an integer for name is also accepted.

If name is a string and is not known, ValueError is raised. If a specific value for name is not supported by the host system, even if it is included in pathconf_names, an OSError is raised with errno.EINVAL for the error number.

Availability: Unix.

os.pathconf_names
Dictionary mapping names accepted by pathconf() and fpathconf() to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system. Availability: Unix.

os.readlink(path)
Return a string representing the path to which the symbolic link points. The result may be either an absolute or relative pathname; if it is relative, it may be converted to an absolute pathname using os.path.join(os.path.dirname(path), result).

Changed in version 2.6: If the path is a Unicode object the result will also be a Unicode object.

Availability: Unix.

os.remove(path)
Remove (delete) the file path. If path is a directory, OSError is raised; see rmdir() below to remove a directory. This is identical to the unlink() function documented below. On Windows, attempting to remove a file that is in use causes an exception to be raised; on Unix, the directory entry is removed but the storage allocated to the file is not made available until the original file is no longer in use.

Availability: Unix, Windows.

os.removedirs(path)
Remove directories recursively. Works like rmdir() except that, if the leaf directory is successfully removed, removedirs() tries to successively remove every parent directory mentioned in path until an error is raised (which is ignored, because it generally means that a parent directory is not empty). For example, os.removedirs('foo/bar/baz') will first remove the directory 'foo/bar/baz', and then remove 'foo/bar' and 'foo' if they are empty. Raises OSError if the leaf directory could not be successfully removed.

New in version 1.5.2.

os.rename(src, dst)
Rename the file or directory src to dst. If dst is a directory, OSError will be raised. On Unix, if dst exists and is a file, it will be replaced silently if the user has permission. The operation may fail on some Unix flavors if src and dst are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement). On Windows, if dst already exists, OSError will be raised even if it is a file; there may be no way to implement an atomic rename when dst names an existing file.

Availability: Unix, Windows.

os.renames(old, new)
Recursive directory or file renaming function. Works like rename(), except creation of any intermediate

directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned away using removedirs().

New in version 1.5.2.

Note: This function can fail with the new directory structure made if you lack permissions needed to remove the leaf directory or file.

---

os.rmdir(path)
Remove (delete) the directory path. Only works when the directory is empty, otherwise, OSError is raised. In order to remove whole directory trees, shutil.rmtree() can be used.

Availability: Unix, Windows.

os.stat(path)
Perform the equivalent of a stat() system call on the given path. (This function follows symlinks; to stat a symlink use lstat().)

The return value is an object whose attributes correspond to the members of the stat structure, namely:

- st_mode - protection bits,
- st_ino - inode number,
- st_dev - device,
- st_nlink - number of hard links,
- st_uid - user id of owner,
- st_gid - group id of owner,
- st_size - size of file, in bytes,
- st_atime - time of most recent access,
- st_mtime - time of most recent content modification,
- st_ctime - platform dependent; time of most recent metadata change on Unix, or the time of creation on Windows)

Changed in version 2.3: If stat_float_times() returns True, the time values are floats, measuring seconds. Fractions of a second may be reported if the system supports that. See stat_float_times() for further discussion.

On some Unix systems (such as Linux), the following attributes may also be available:

- st_blocks - number of 512-byte blocks allocated for file
- st_blksize - filesystem blocksize for efficient file system I/O
- st_rdev - type of device if an inode device
- st_flags - user defined flags for file

On other Unix systems (such as FreeBSD), the following attributes may be available (but may be only filled out if root tries to use them):

- st_gen - file generation number
- st_birthtime - time of file creation

On RISCOS systems, the following attributes are also available:

- st_ftype (file type)

- st_attrs (attributes)
- st_obtype (object type).

Note: The exact meaning and resolution of the st_atime, st_mtime, and st_ctime attributes depend on the operating system and the file system. For example, on Windows systems using the FAT or FAT32 file systems, st_mtime has 2-second resolution, and st_atime has only 1-day resolution. See your operating system documentation for details.

For backward compatibility, the return value of stat() is also accessible as a tuple of at least 10 integers giving the most important (and portable) members of the stat structure, in the order st_mode, st_ino, st_dev, st_nlink, st_uid, st_gid, st_size, st_atime, st_mtime, st_ctime. More items may be added at the end by some implementations.

The standard module stat defines functions and constants that are useful for extracting information from a stat structure. (On Windows, some items are filled with dummy values.)

Example:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
(33188, 422511, 769, 1, 1032, 100, 926, 1105022698,1105022732, 1105022732)
>>> statinfo.st_size
926
```

Availability: Unix, Windows.

Changed in version 2.2: Added access to values as attributes of the returned object.

Changed in version 2.5: Added st_gen and st_birthtime.

os.stat_float_times([newvalue])
Determine whether stat_result represents time stamps as float objects. If newvalue is True, future calls to stat() return floats, if it is False, future calls return ints. If newvalue is omitted, return the current setting.

For compatibility with older Python versions, accessing stat_result as a tuple always returns integers.

Changed in version 2.5: Python now returns float values by default. Applications which do not work correctly with floating point time stamps can use this function to restore the old behaviour.

The resolution of the timestamps (that is the smallest possible fraction) depends on the system. Some systems only support second resolution; on these systems, the fraction will always be zero.

It is recommended that this setting is only changed at program startup time in the __main__ module; libraries should never change this setting. If an application uses a library that works incorrectly if floating point time stamps are processed, this application should turn the feature off until the library has been corrected.

os.statvfs(path)
Perform a statvfs() system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the statvfs structure, namely: f_bsize, f_frsize, f_blocks, f_bfree, f_bavail, f_files, f_ffree, f_favail, f_flag, f_namemax.

For backward compatibility, the return value is also accessible as a tuple whose values correspond to the attributes, in the order given above. The standard module statvfs defines constants that are useful for extracting information from a statvfs structure when accessing it as a sequence; this remains useful

when writing code that needs to work with versions of Python that don't support accessing the fields as attributes.

Availability: Unix.

Changed in version 2.2: Added access to values as attributes of the returned object.

os.symlink(source, link_name)
Create a symbolic link pointing to source named link_name.

Availability: Unix.

os.tempnam([dir[, prefix]])
Return a unique path name that is reasonable for creating a temporary file. This will be an absolute path that names a potential directory entry in the directory dir or a common location for temporary files if dir is omitted or None. If given and not None, prefix is used to provide a short prefix to the filename. Applications are responsible for properly creating and managing files created using paths returned by tempnam(); no automatic cleanup is provided. On Unix, the environment variable TMPDIR overrides dir, while on Windows TMP is used. The specific behavior of this function depends on the C library implementation; some aspects are underspecified in system documentation.

> Warning: Use of tempnam() is vulnerable to symlink attacks; consider using tmpfile() (section File Object Creation) instead.

Availability: Unix, Windows.

os.tmpnam()
Return a unique path name that is reasonable for creating a temporary file. This will be an absolute path that names a potential directory entry in a common location for temporary files. Applications are responsible for properly creating and managing files created using paths returned by tmpnam(); no automatic cleanup is provided.

> Warning: Use of tmpnam() is vulnerable to symlink attacks; consider using tmpfile() (section File Object Creation) instead.

Availability: Unix, Windows. This function probably shouldn't be used on Windows, though: Microsoft's implementation of tmpnam() always creates a name in the root directory of the current drive, and that's generally a poor location for a temp file (depending on privileges, you may not even be able to open a file using this name).

os.TMP_MAX
The maximum number of unique names that tmpnam() will generate before reusing names.

os.unlink(path)
Remove (delete) the file path. This is the same function as remove(); the unlink() name is its traditional Unix name.

Availability: Unix, Windows.

os.utime(path, times)
Set the access and modified times of the file specified by path. If times is None, then the file's access and modified times are set to the current time. (The effect is similar to running the Unix program touch on the path.) Otherwise, times must be a 2-tuple of numbers, of the form (atime, mtime) which is used to set the access and modified times, respectively. Whether a directory can be given for path depends on whether the operating system implements directories as files (for example, Windows does

not). Note that the exact times you set here may not be returned by a subsequent stat() call, depending on the resolution with which your operating system records access and modification times; see stat().

Changed in version 2.0: Added support for None for times.

Availability: Unix, Windows.

os.walk(top, topdown=True, onerror=None, followlinks=False)

Generate the file names in a directory tree by walking the tree either top-down or bottom-up. For each directory in the tree rooted at directory top (including top itself), it yields a 3-tuple (dirpath, dirnames, filenames).

dirpath is a string, the path to the directory. dirnames is a list of the names of the subdirectories in dirpath (excluding '.' and '..'). filenames is a list of the names of the non-directory files in dirpath. Note that the names in the lists contain no path components. To get a full path (which begins with top) to a file or directory in dirpath, do os.path.join(dirpath, name).

If optional argument topdown is True or not specified, the triple for a directory is generated before the triples for any of its subdirectories (directories are generated top-down). If topdown is False, the triple for a directory is generated after the triples for all of its subdirectories (directories are generated bottom-up). No matter the value of topdown, the list of subdirectories is retrieved before the tuples for the directory and its subdirectories are generated.

When topdown is True, the caller can modify the dirnames list in-place (perhaps using del or slice assignment), and walk() will only recurse into the subdirectories whose names remain in dirnames; this can be used to prune the search, impose a specific order of visiting, or even to inform walk() about directories the caller creates or renames before it resumes walk() again. Modifying dirnames when topdown is False has no effect on the behavior of the walk, because in bottom-up mode the directories in dirnames are generated before dirpath itself is generated.

By default, errors from the listdir() call are ignored. If optional argument onerror is specified, it should be a function; it will be called with one argument, an OSError instance. It can report the error to continue with the walk, or raise the exception to abort the walk. Note that the filename is available as the filename attribute of the exception object.

By default, walk() will not walk down into symbolic links that resolve to directories. Set followlinks to True to visit directories pointed to by symlinks, on systems that support them.

New in version 2.6: The followlinks parameter.

Note: Be aware that setting followlinks to True can lead to infinite recursion if a link points to a parent directory of itself. walk() does not keep track of the directories it visited already.

Note: If you pass a relative pathname, don't change the current working directory between resumptions of walk(). walk() never changes the current directory, and assumes that its caller doesn't either.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn't look under any CVS subdirectory:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print root, "consumes",
    print sum(getsize(join(root, name)) for name in files),
    print "bytes in", len(files), "non-directory files"
```

(continues on next page)

(continued from previous page)

```
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

In the next example, walking the tree bottom-up is essential: rmdir() doesn't allow deleting a directory before the directory is empty:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

New in version 2.3.

15.1.5 Process Management

These functions may be used to create and manage processes.

The various exec* functions take a list of arguments for the new program loaded into the process. In each case, the first of these arguments is passed to the new program as its own name rather than as an argument a user may have typed on a command line. For the C programmer, this is the argv[0] passed to a program's main(). For example, os.execv('/bin/echo', ['foo', 'bar']) will only print bar on standard output; foo will seem to be ignored.

os.abort()
: Generate a SIGABRT signal to the current process. On Unix, the default behavior is to produce a core dump; on Windows, the process immediately returns an exit code of 3. Be aware that calling this function will not call the Python signal handler registered for SIGABRT with signal.signal().

  Availability: Unix, Windows.

os.execl(path, arg0, arg1, ...)
os.execle(path, arg0, arg1, ..., env)
os.execlp(file, arg0, arg1, ...)
os.execlpe(file, arg0, arg1, ..., env)
os.execv(path, args)
os.execve(path, args, env)
os.execvp(file, args)
os.execvpe(file, args, env)
: These functions all execute a new program, replacing the current process; they do not return. On Unix, the new executable is loaded into the current process, and will have the same process id as the caller. Errors will be reported as OSError exceptions.

  The current process is replaced immediately. Open file objects and descriptors are not flushed, so if there may be data buffered on these open files, you should flush them using sys.stdout.flush() or os.fsync() before calling an exec* function.

  The "l" and "v" variants of the exec* functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the execl*() functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed

in a list or tuple as the args parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced.

The variants which include a "p" near the end (execlp(), execlpe(), execvp(), and execvpe()) will use the PATH environment variable to locate the program file. When the environment is being replaced (using one of the exec*e variants, discussed in the next paragraph), the new environment is used as the source of the PATH variable. The other variants, execl(), execle(), execv(), and execve(), will not use the PATH variable to locate the executable; path must contain an appropriate absolute or relative path.

For execle(), execlpe(), execve(), and execvpe() (note that these all end in "e"), the env parameter must be a mapping which is used to define the environment variables for the new process (these are used instead of the current process' environment); the functions execl(), execlp(), execv(), and execvp() all cause the new process to inherit the environment of the current process.

Availability: Unix, Windows.

os._exit(n)
Exit the process with status n, without calling cleanup handlers, flushing stdio buffers, etc.

Availability: Unix, Windows.

Note: The standard way to exit is sys.exit(n). _exit() should normally only be used in the child process after a fork().

The following exit codes are defined and can be used with _exit(), although they are not required. These are typically used for system programs written in Python, such as a mail server's external command delivery program.

Note: Some of these may not be available on all Unix platforms, since there is some variation. These constants are defined where they are defined by the underlying platform.

---

os.EX_OK
Exit code that means no error occurred.

Availability: Unix.

New in version 2.3.

os.EX_USAGE
Exit code that means the command was used incorrectly, such as when the wrong number of arguments are given.

Availability: Unix.

New in version 2.3.

os.EX_DATAERR
Exit code that means the input data was incorrect.

Availability: Unix.

New in version 2.3.

os.EX_NOINPUT
Exit code that means an input file did not exist or was not readable.

Availability: Unix.

New in version 2.3.

os.EX_NOUSER
Exit code that means a specified user did not exist.

Availability: Unix.

New in version 2.3.

os.EX_NOHOST
Exit code that means a specified host did not exist.

Availability: Unix.

New in version 2.3.

os.EX_UNAVAILABLE
Exit code that means that a required service is unavailable.

Availability: Unix.

New in version 2.3.

os.EX_SOFTWARE
Exit code that means an internal software error was detected.

Availability: Unix.

New in version 2.3.

os.EX_OSERR
Exit code that means an operating system error was detected, such as the inability to fork or create a pipe.

Availability: Unix.

New in version 2.3.

os.EX_OSFILE
Exit code that means some system file did not exist, could not be opened, or had some other kind of error.

Availability: Unix.

New in version 2.3.

os.EX_CANTCREAT
Exit code that means a user specified output file could not be created.

Availability: Unix.

New in version 2.3.

os.EX_IOERR
Exit code that means that an error occurred while doing I/O on some file.

Availability: Unix.

New in version 2.3.

os.EX_TEMPFAIL
Exit code that means a temporary failure occurred. This indicates something that may not really be an error, such as a network connection that couldn't be made during a retryable operation.

Availability: Unix.

New in version 2.3.

os.EX_PROTOCOL
Exit code that means that a protocol exchange was illegal, invalid, or not understood.

Availability: Unix.

New in version 2.3.

os.EX_NOPERM
Exit code that means that there were insufficient permissions to perform the operation (but not intended for file system problems).

Availability: Unix.

New in version 2.3.

os.EX_CONFIG
Exit code that means that some kind of configuration error occurred.

Availability: Unix.

New in version 2.3.

os.EX_NOTFOUND
Exit code that means something like "an entry was not found".

Availability: Unix.

New in version 2.3.

os.fork()
Fork a child process. Return 0 in the child and the child's process id in the parent. If an error occurs OSError is raised.

Note that some platforms including FreeBSD <= 6.3, Cygwin and OS/2 EMX have known issues when using fork() from a thread.

Warning: See ssl for applications that use the SSL module with fork().

Availability: Unix.

os.forkpty()
Fork a child process, using a new pseudo-terminal as the child's controlling terminal. Return a pair of (pid, fd), where pid is 0 in the child, the new child's process id in the parent, and fd is the file descriptor of the master end of the pseudo-terminal. For a more portable approach, use the pty module. If an error occurs OSError is raised.

Availability: some flavors of Unix.

os.kill(pid, sig)
Send signal sig to the process pid. Constants for the specific signals available on the host platform are defined in the signal module.

Windows: The signal.CTRL_C_EVENT and signal.CTRL_BREAK_EVENT signals are special signals which can only be sent to console processes which share a common console window, e.g., some subprocesses. Any other value for sig will cause the process to be unconditionally killed by the TerminateProcess API, and the exit code will be set to sig. The Windows version of kill() additionally takes process handles to be killed.

New in version 2.7: Windows support

os.killpg(pgid, sig)
Send the signal sig to the process group pgid.

Availability: Unix.

New in version 2.3.

os.nice(increment)
Add increment to the process's "niceness". Return the new niceness.

Availability: Unix.

os.plock(op)
Lock program segments into memory. The value of op (defined in <sys/lock.h>) determines which segments are locked.

Availability: Unix.

os.popen(...)
os.popen2(...)
os.popen3(...)
os.popen4(...)
Run child processes, returning opened pipes for communications. These functions are described in section File Object Creation.

os.spawnl(mode, path, ...)
os.spawnle(mode, path, ..., env)
os.spawnlp(mode, file, ...)
os.spawnlpe(mode, file, ..., env)
os.spawnv(mode, path, args)
os.spawnve(mode, path, args, env)
os.spawnvp(mode, file, args)
os.spawnvpe(mode, file, args, env)
Execute the program path in a new process.

(Note that the subprocess module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using these functions. Check especially the Replacing Older Functions with the subprocess Module section.)

If mode is P_NOWAIT, this function returns the process id of the new process; if mode is P_WAIT, returns the process's exit code if it exits normally, or -signal, where signal is the signal that killed the process. On Windows, the process id will actually be the process handle, so can be used with the waitpid() function.

The "l" and "v" variants of the spawn* functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the spawnl*() functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the args parameter. In either case, the arguments to the child process must start with the name of the command being run.

The variants which include a second "p" near the end (spawnlp(), spawnlpe(), spawnvp(), and spawnvpe()) will use the PATH environment variable to locate the program file. When the environment is being replaced (using one of the spawn*e variants, discussed in the next paragraph), the new environment is used as the source of the PATH variable. The other variants, spawnl(), spawnle(), spawnv(), and spawnve(), will not use the PATH variable to locate the executable; path must contain an appropriate absolute or relative path.

For spawnle(), spawnlpe(), spawnve(), and spawnvpe() (note that these all end in "e"), the env parameter must be a mapping which is used to define the environment variables for the new process (they are used instead of the current process' environment); the functions spawnl(), spawnlp(), spawnv(), and spawnvp() all cause the new process to inherit the environment of the current process. Note that

keys and values in the env dictionary must be strings; invalid keys or values will cause the function to fail, with a return value of 127.

As an example, the following calls to spawnlp() and spawnvpe() are equivalent:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

Availability: Unix, Windows. spawnlp(), spawnlpe(), spawnvp() and spawnvpe() are not available on Windows. spawnle() and spawnve() are not thread-safe on Windows; we advise you to use the subprocess module instead.

New in version 1.6.

os.P_NOWAIT
os.P_NOWAITO
Possible values for the mode parameter to the spawn* family of functions. If either of these values is given, the spawn*() functions will return as soon as the new process has been created, with the process id as the return value.

Availability: Unix, Windows.

New in version 1.6.

os.P_WAIT
Possible value for the mode parameter to the spawn* family of functions. If this is given as mode, the spawn*() functions will not return until the new process has run to completion and will return the exit code of the process the run is successful, or -signal if a signal kills the process.

Availability: Unix, Windows.

New in version 1.6.

os.P_DETACH
os.P_OVERLAY
Possible values for the mode parameter to the spawn* family of functions. These are less portable than those listed above. P_DETACH is similar to P_NOWAIT, but the new process is detached from the console of the calling process. If P_OVERLAY is used, the current process will be replaced; the spawn*() function will not return.

Availability: Windows.

New in version 1.6.

os.startfile(path[, operation])
Start a file with its associated application.

When operation is not specified or 'open', this acts like double-clicking the file in Windows Explorer, or giving the file name as an argument to the start command from the interactive command shell: the file is opened with whatever application (if any) its extension is associated.

When another operation is given, it must be a "command verb" that specifies what should be done with the file. Common verbs documented by Microsoft are 'print' and 'edit' (to be used on files) as well as 'explore' and 'find' (to be used on directories).

startfile() returns as soon as the associated application is launched. There is no option to wait for the application to close, and no way to retrieve the application's exit status. The path parameter is relative to the current directory. If you want to use an absolute path, make sure the first character is not a slash

('/'); the underlying Win32 ShellExecute() function doesn't work if it is. Use the os.path.normpath() function to ensure that the path is properly encoded for Win32.

Availability: Windows.

New in version 2.0.

New in version 2.5: The operation parameter.

os.system(command)
Execute the command (a string) in a subshell. This is implemented by calling the Standard C function system(), and has the same limitations. Changes to sys.stdin, etc. are not reflected in the environment of the executed command.

On Unix, the return value is the exit status of the process encoded in the format specified for wait(). Note that POSIX does not specify the meaning of the return value of the C system() function, so the return value of the Python function is system-dependent.

On Windows, the return value is that returned by the system shell after running command, given by the Windows environment variable COMSPEC: on command.com systems (Windows 95, 98 and ME) this is always 0; on cmd.exe systems (Windows NT, 2000 and XP) this is the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

The subprocess module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using this function. See the Replacing Older Functions with the subprocess Module section in the subprocess documentation for some helpful recipes.

Availability: Unix, Windows.

os.times()
Return a 5-tuple of floating point numbers indicating accumulated (processor or other) times, in seconds. The items are: user time, system time, children's user time, children's system time, and elapsed real time since a fixed point in the past, in that order. See the Unix manual page times(2) or the corresponding Windows Platform API documentation. On Windows, only the first two items are filled, the others are zero.

Availability: Unix, Windows

os.wait()
Wait for completion of a child process, and return a tuple containing its pid and exit status indication: a 16-bit number, whose low byte is the signal number that killed the process, and whose high byte is the exit status (if the signal number is zero); the high bit of the low byte is set if a core file was produced.

Availability: Unix.

os.waitpid(pid, options)
The details of this function differ on Unix and Windows.

On Unix: Wait for completion of a child process given by process id pid, and return a tuple containing its process id and exit status indication (encoded as for wait()). The semantics of the call are affected by the value of the integer options, which should be 0 for normal operation.

If pid is greater than 0, waitpid() requests status information for that specific process. If pid is 0, the request is for the status of any child in the process group of the current process. If pid is -1, the request pertains to any child of the current process. If pid is less than -1, status is requested for any process in the process group -pid (the absolute value of pid).

An OSError is raised with the value of errno when the syscall returns -1.

On Windows: Wait for completion of a process given by process handle pid, and return a tuple containing pid, and its exit status shifted left by 8 bits (shifting makes cross-platform use of the

function easier). A pid less than or equal to 0 has no special meaning on Windows, and raises an exception. The value of integer options has no effect. pid can refer to any process whose id is known, not necessarily a child process. The spawn* functions called with P_NOWAIT return suitable process handles.

os.wait3(options)
Similar to waitpid(), except no process id argument is given and a 3-element tuple containing the child's process id, exit status indication, and resource usage information is returned. Refer to resource.getrusage() for details on resource usage information. The option argument is the same as that provided to waitpid() and wait4().

Availability: Unix.

New in version 2.5.

os.wait4(pid, options)
Similar to waitpid(), except a 3-element tuple, containing the child's process id, exit status indication, and resource usage information is returned. Refer to resource.getrusage() for details on resource usage information. The arguments to wait4() are the same as those provided to waitpid().

Availability: Unix.

New in version 2.5.

os.WNOHANG
The option for waitpid() to return immediately if no child process status is available immediately. The function returns (0, 0) in this case.

Availability: Unix.

os.WCONTINUED
This option causes child processes to be reported if they have been continued from a job control stop since their status was last reported.

Availability: Some Unix systems.

New in version 2.3.

os.WUNTRACED
This option causes child processes to be reported if they have been stopped but their current state has not been reported since they were stopped.

Availability: Unix.

New in version 2.3.

The following functions take a process status code as returned by system(), wait(), or waitpid() as a parameter. They may be used to determine the disposition of a process.

os.WCOREDUMP(status)
Return True if a core dump was generated for the process, otherwise return False.

Availability: Unix.

New in version 2.3.

os.WIFCONTINUED(status)
Return True if the process has been continued from a job control stop, otherwise return False.

Availability: Unix.

New in version 2.3.

os.WIFSTOPPED(status)
Return True if the process has been stopped, otherwise return False.

Availability: Unix.

os.WIFSIGNALED(status)
Return True if the process exited due to a signal, otherwise return False.

Availability: Unix.

os.WIFEXITED(status)
Return True if the process exited using the exit(2) system call, otherwise return False.

Availability: Unix.

os.WEXITSTATUS(status)
If WIFEXITED(status) is true, return the integer parameter to the exit(2) system call. Otherwise, the return value is meaningless.

Availability: Unix.

os.WSTOPSIG(status)
Return the signal which caused the process to stop.

Availability: Unix.

os.WTERMSIG(status)
Return the signal which caused the process to exit.

Availability: Unix.

15.1.6 Miscellaneous System Information

os.confstr(name)
Return string-valued system configuration values. name specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given as the keys of the confstr_names dictionary. For configuration variables not included in that mapping, passing an integer for name is also accepted.

If the configuration value specified by name isn't defined, None is returned.

If name is a string and is not known, ValueError is raised. If a specific value for name is not supported by the host system, even if it is included in confstr_names, an OSError is raised with errno.EINVAL for the error number.

Availability: Unix

os.confstr_names
Dictionary mapping names accepted by confstr() to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

os.getloadavg()
Return the number of processes in the system run queue averaged over the last 1, 5, and 15 minutes or raises OSError if the load average was unobtainable.

Availability: Unix.

New in version 2.3.

os.sysconf(name)
Return integer-valued system configuration values. If the configuration value specified by name isn't defined, -1 is returned. The comments regarding the name parameter for confstr() apply here as well; the dictionary that provides information on the known names is given by sysconf_names.

Availability: Unix.

os.sysconf_names
Dictionary mapping names accepted by sysconf() to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

The following data values are used to support path manipulation operations. These are defined for all platforms.

Higher-level operations on pathnames are defined in the os.path module.

os.curdir
The constant string used by the operating system to refer to the current directory. This is '.' for Windows and POSIX. Also available via os.path.

os.pardir
The constant string used by the operating system to refer to the parent directory. This is '..' for Windows and POSIX. Also available via os.path.

os.sep
The character used by the operating system to separate pathname components. This is '/' for POSIX and '\\' for Windows. Note that knowing this is not sufficient to be able to parse or concatenate pathnames — use os.path.split() and os.path.join() — but it is occasionally useful. Also available via os.path.

os.altsep
An alternative character used by the operating system to separate pathname components, or None if only one separator character exists. This is set to '/' on Windows systems where sep is a backslash. Also available via os.path.

os.extsep
The character which separates the base filename from the extension; for example, the '.' in os.py. Also available via os.path.

New in version 2.2.

os.pathsep
The character conventionally used by the operating system to separate search path components (as in PATH), such as ':' for POSIX or ';' for Windows. Also available via os.path.

os.defpath
The default search path used by exec*p* and spawn*p* if the environment doesn't have a 'PATH' key. Also available via os.path.

os.linesep
The string used to separate (or, rather, terminate) lines on the current platform. This may be a single character, such as '\n' for POSIX, or multiple characters, for example, '\r\n' for Windows. Do not use os.linesep as a line terminator when writing files opened in text mode (the default); use a single '\n' instead, on all platforms.

os.devnull
The file path of the null device. For example: '/dev/null' for POSIX, 'nul' for Windows. Also available via os.path.

New in version 2.4.

15.1.7 Miscellaneous Functions

os.urandom(n)
Return a string of n random bytes suitable for cryptographic use.

This function returns random bytes from an OS-specific randomness source. The returned data should be unpredictable enough for cryptographic applications, though its exact quality depends on the OS implementation. On a UNIX-like system this will query /dev/urandom, and on Windows it will use CryptGenRandom(). If a randomness source is not found, NotImplementedError will be raised.

For an easy-to-use interface to the random number generator provided by your platform, please see random.SystemRandom.

New in version 2.4.

15.2 io — Core tools for working with streams

New in version 2.6.

The io module provides the Python interfaces to stream handling. Under Python 2.x, this is proposed as an alternative to the built-in file object, but in Python 3.x it is the default interface to access files and streams.

Note: Since this module has been designed primarily for Python 3.x, you have to be aware that all uses of "bytes" in this document refer to the str type (of which bytes is an alias), and all uses of "text" refer to the unicode type. Furthermore, those two types are not interchangeable in the io APIs.

At the top of the I/O hierarchy is the abstract base class IOBase. It defines the basic interface to a stream. Note, however, that there is no separation between reading and writing to streams; implementations are allowed to raise an IOError if they do not support a given operation.

Extending IOBase is RawIOBase which deals simply with the reading and writing of raw bytes to a stream. FileIO subclasses RawIOBase to provide an interface to files in the machine's file system.

BufferedIOBase deals with buffering on a raw byte stream (RawIOBase). Its subclasses, BufferedWriter, BufferedReader, and BufferedRWPair buffer streams that are readable, writable, and both readable and writable. BufferedRandom provides a buffered interface to random access streams. BytesIO is a simple stream of in-memory bytes.

Another IOBase subclass, TextIOBase, deals with streams whose bytes represent text, and handles encoding and decoding from and to unicode strings. TextIOWrapper, which extends it, is a buffered text interface to a buffered raw stream (BufferedIOBase). Finally, StringIO is an in-memory stream for unicode text.

Argument names are not part of the specification, and only the arguments of open() are intended to be used as keyword arguments.

15.2.1 Module Interface

io.DEFAULT_BUFFER_SIZE
An int containing the default buffer size used by the module's buffered I/O classes. open() uses the file's blksize (as obtained by os.stat()) if possible.

io.open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True)
Open file and return a corresponding stream. If the file cannot be opened, an IOError is raised.

file is either a string giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless closefd is set to False.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'w'` for writing (truncating the file if it already exists), and `'a'` for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'a'`	open for writing, appending to the end of the file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open a disk file for updating (reading and writing)
`'U'`	universal newlines mode (for backwards compatibility; should not be used in new code)

The default mode is `'rt'` (open for reading text). For binary random access, the mode `'w+b'` opens and truncates the file to 0 bytes, while `'r+b'` opens the file without truncation.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (including `'b'` in the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when `'t'` is included in the mode argument), the contents of the file are returned as unicode strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on DEFAULT_BUFFER_SIZE. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "Interactive" text files (files for which isatty() returns True) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever locale.getpreferredencoding() returns), but any encoding supported by Python can be used. See the codecs module for the list of supported encodings.

errors is an optional string that specifies how encoding and decoding errors are to be handled—this cannot be used in binary mode. Pass `'strict'` to raise a ValueError exception if there is an encoding error (the default of None has the same effect), or pass `'ignore'` to ignore errors. (Note that ignoring encoding errors can lead to data loss.) `'replace'` causes a replacement marker (such as `'?'`) to be inserted where there is malformed data. When writing, `'xmlcharrefreplace'` (replace with the appropriate XML character reference) or `'backslashreplace'` (replace with backslashed escape sequences) can be used. Any other error handling name that has been registered with codecs.register_error() is also valid.

newline controls how universal newlines works (it only applies to text mode). It can be None, '', '\n', '\r', and '\r\n'. It works as follows:

- On input, if newline is None, universal newlines mode is enabled. Lines in the input can end in '\n', '\r', or '\r\n', and these are translated into '\n' before being returned to the caller. If it is '', universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- On output, if newline is None, any '\n' characters written are translated to the system default line separator, os.linesep. If newline is '', no translation takes place. If newline is any of the other legal values, any '\n' characters written are translated to the given string.

If closefd is False and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given closefd has no effect and must be True (the default).

The type of file object returned by the open() function depends on the mode. When open() is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a subclass of TextIOBase (specifically TextIOWrapper). When used to open a file in a binary mode with buffering, the returned class is a subclass of BufferedIOBase. The exact class varies: in read binary mode, it returns a BufferedReader; in write binary and append binary modes, it returns a BufferedWriter, and in read/write mode, it returns a BufferedRandom. When buffering is disabled, the raw stream, a subclass of RawIOBase, FileIO, is returned.

It is also possible to use an unicode or bytes string as a file for both reading and writing. For unicode strings StringIO can be used like a file opened in text mode, and for bytes a BytesIO can be used like a file opened in a binary mode.

exception io.BlockingIOError
Error raised when blocking would occur on a non-blocking stream. It inherits IOError.

In addition to those of IOError, BlockingIOError has one attribute:

characters_written
An integer containing the number of characters written to the stream before it blocked.

exception io.UnsupportedOperation
An exception inheriting IOError and ValueError that is raised when an unsupported operation is called on a stream.

15.2.2 I/O Base Classes

class io.IOBase
The abstract base class for all I/O classes, acting on streams of bytes. There is no public constructor.

This class provides empty abstract implementations for many methods that derived classes can override selectively; the default implementations represent a file that cannot be read, written or seeked.

Even though IOBase does not declare read(), readinto(), or write() because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise an IOError when operations they do not support are called.

The basic type used for binary data read from or written to a file is bytes (also known as str). Method arguments may also be bytearray or memoryview of arrays of bytes. In some cases, such as readinto(), a writable object such as bytearray is required. Text I/O classes work with unicode data.

Changed in version 2.7: Implementations should support memoryview arguments.

Note that calling any method (even inquiries) on a closed stream is undefined. Implementations may raise IOError in this case.

IOBase (and its subclasses) support the iterator protocol, meaning that an IOBase object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a text stream (yielding unicode strings). See readline() below.

IOBase is also a context manager and therefore supports the with statement. In this example, file is closed after the with statement's suite is finished—even if an exception occurs:

```
with io.open('spam.txt', 'w') as file:
    file.write(u'Spam and eggs!')
```

IOBase provides these data attributes and methods:

close()
: Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a ValueError.

  As a convenience, it is allowed to call this method more than once; only the first call, however, will have an effect.

closed
: True if the stream is closed.

fileno()
: Return the underlying file descriptor (an integer) of the stream if it exists. An IOError is raised if the IO object does not use a file descriptor.

flush()
: Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.

isatty()
: Return True if the stream is interactive (i.e., connected to a terminal/tty device).

readable()
: Return True if the stream can be read from. If False, read() will raise IOError.

readline(limit=-1)
: Read and return one line from the stream. If limit is specified, at most limit bytes will be read.

  The line terminator is always `b'\n'` for binary files; for text files, the newline argument to open() can be used to select the line terminator(s) recognized.

readlines(hint=-1)
: Read and return a list of lines from the stream. hint can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds hint.

  Note that it's already possible to iterate on file objects using for line in file: ... without calling file.readlines().

seek(offset[, whence])
: Change the stream position to the given byte offset. offset is interpreted relative to the position indicated by whence. The default value for whence is SEEK_SET. Values for whence are:

  - SEEK_SET or 0 – start of the stream (the default); offset should be zero or positive
  - SEEK_CUR or 1 – current stream position; offset may be negative
  - SEEK_END or 2 – end of the stream; offset is usually negative

Return the new absolute position.

New in version 2.7: The SEEK_* constants

seekable()
Return True if the stream supports random access. If False, seek(), tell() and truncate() will raise IOError.

tell()
Return the current stream position.

truncate(size=None)
Resize the stream to the given size in bytes (or the current position if size is not specified). The current stream position isn't changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled, on Windows they're undetermined). The new file size is returned.

writable()
Return True if the stream supports writing. If False, write() and truncate() will raise IOError.

writelines(lines)
Write a list of lines to the stream. Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

__del__()
Prepare for object destruction. IOBase provides a default implementation of this method that calls the instance's close() method.

class io.RawIOBase
Base class for raw binary I/O. It inherits IOBase. There is no public constructor.

Raw binary I/O typically provides low-level access to an underlying OS device or API, and does not try to encapsulate it in high-level primitives (this is left to Buffered I/O and Text I/O, described later in this page).

In addition to the attributes and methods from IOBase, RawIOBase provides the following methods:

read(n=-1)
Read up to n bytes from the object and return them. As a convenience, if n is unspecified or -1, readall() is called. Otherwise, only one system call is ever made. Fewer than n bytes may be returned if the operating system call returns fewer than n bytes.

If 0 bytes are returned, and n was not 0, this indicates end of file. If the object is in non-blocking mode and no bytes are available, None is returned.

readall()
Read and return all the bytes from the stream until EOF, using multiple calls to the stream if necessary.

readinto(b)
Read up to len(b) bytes into b, and return the number of bytes read. The object b should be a pre-allocated, writable array of bytes, either bytearray or memoryview. If the object is in non-blocking mode and no bytes are available, None is returned.

write(b)
Write b to the underlying raw stream, and return the number of bytes written. The object b should be an array of bytes, either bytes, bytearray, or memoryview. The return value can be less than len(b), depending on specifics of the underlying raw stream, and especially if it is in non-blocking mode. None is returned if the raw stream is set not to block and no single byte could be readily written to it. The caller may release or mutate b after this method returns, so the implementation should only access b during the method call.

class io.BufferedIOBase

Base class for binary streams that support some kind of buffering. It inherits IOBase. There is no public constructor.

The main difference with RawIOBase is that methods read(), readinto() and write() will try (respectively) to read as much input as requested or to consume all given output, at the expense of making perhaps more than one system call.

In addition, those methods can raise BlockingIOError if the underlying raw stream is in non-blocking mode and cannot take or give enough data; unlike their RawIOBase counterparts, they will never return None.

Besides, the read() method does not have a default implementation that defers to readinto().

A typical BufferedIOBase implementation should not inherit from a RawIOBase implementation, but wrap one, like BufferedWriter and BufferedReader do.

BufferedIOBase provides or overrides these methods and attribute in addition to those from IOBase:

raw

The underlying raw stream (a RawIOBase instance) that BufferedIOBase deals with. This is not part of the BufferedIOBase API and may not exist on some implementations.

detach()

Separate the underlying raw stream from the buffer and return it.

After the raw stream has been detached, the buffer is in an unusable state.

Some buffers, like BytesIO, do not have the concept of a single raw stream to return from this method. They raise UnsupportedOperation.

New in version 2.7.

read(n=-1)

Read and return up to n bytes. If the argument is omitted, None, or negative, data is read and returned until EOF is reached. An empty bytes object is returned if the stream is already at EOF.

If the argument is positive, and the underlying raw stream is not interactive, multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams, at most one raw read will be issued, and a short result does not imply that EOF is imminent.

A BlockingIOError is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

read1(n=-1)

Read and return up to n bytes, with at most one call to the underlying raw stream's read() method. This can be useful if you are implementing your own buffering on top of a BufferedIOBase object.

readinto(b)

Read up to len(b) bytes into b, and return the number of bytes read. The object b should be a pre-allocated, writable array of bytes, either bytearray or memoryview.

Like read(), multiple reads may be issued to the underlying raw stream, unless the latter is 'interactive'.

A BlockingIOError is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

write(b)

Write b, and return the number of bytes written (always equal to len(b), since if the write fails an IOError will be raised). The object b should be an array of bytes, either bytes, bytearray,

or memoryview. Depending on the actual implementation, these bytes may be readily written to the underlying stream, or held in a buffer for performance and latency reasons.

When in non-blocking mode, a BlockingIOError is raised if the data needed to be written to the raw stream but it couldn't accept all the data without blocking.

The caller may release or mutate b after this method returns, so the implementation should only access b during the method call.

15.2.3 Raw File I/O

class io.FileIO(name, mode='r', closefd=True)

FileIO represents an OS-level file containing bytes data. It implements the RawIOBase interface (and therefore the IOBase interface, too).

The name can be one of two things:

- a string representing the path to the file which will be opened;
- an integer representing the number of an existing OS-level file descriptor to which the resulting FileIO object will give access.

The mode can be `'r'`, `'w'` or `'a'` for reading (default), writing, or appending. The file will be created if it doesn't exist when opened for writing or appending; it will be truncated when opened for writing. Add a `'+'` to the mode to allow simultaneous reading and writing.

The read() (when called with a positive argument), readinto() and write() methods on this class will only make one system call.

In addition to the attributes and methods from IOBase and RawIOBase, FileIO provides the following data attributes and methods:

mode
: The mode as given in the constructor.

name
: The file name. This is the file descriptor of the file when no name is given in the constructor.

15.2.4 Buffered Streams

Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does.

class io.BytesIO([initial_bytes])

A stream implementation using an in-memory bytes buffer. It inherits BufferedIOBase.

The optional argument initial_bytes is a bytes object that contains initial data.

BytesIO provides or overrides these methods in addition to those from BufferedIOBase and IOBase:

getvalue()
: Return bytes containing the entire contents of the buffer.

read1()
: In BytesIO, this is the same as read().

class io.BufferedReader(raw, buffer_size=DEFAULT_BUFFER_SIZE)

A buffer providing higher-level access to a readable, sequential RawIOBase object. It inherits BufferedIOBase. When reading data from this object, a larger amount of data may be requested from the underlying raw stream, and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

The constructor creates a BufferedReader for the given readable raw stream and buffer_size. If buffer_size is omitted, DEFAULT_BUFFER_SIZE is used.

BufferedReader provides or overrides these methods in addition to those from BufferedIOBase and IOBase:

peek([n])
: Return bytes from the stream without advancing the position. At most one single read on the raw stream is done to satisfy the call. The number of bytes returned may be less or more than requested.

read([n])
: Read and return n bytes, or if n is not given or negative, until EOF or if the read call would block in non-blocking mode.

read1(n)
: Read and return up to n bytes with only one call on the raw stream. If at least one byte is buffered, only buffered bytes are returned. Otherwise, one raw stream read call is made.

class io.BufferedWriter(raw, buffer_size=DEFAULT_BUFFER_SIZE)
: A buffer providing higher-level access to a writeable, sequential RawIOBase object. It inherits BufferedIOBase. When writing to this object, data is normally held into an internal buffer. The buffer will be written out to the underlying RawIOBase object under various conditions, including:

  - when the buffer gets too small for all pending data;
  - when flush() is called;
  - when a seek() is requested (for BufferedRandom objects);
  - when the BufferedWriter object is closed or destroyed.

  The constructor creates a BufferedWriter for the given writeable raw stream. If the buffer_size is not given, it defaults to DEFAULT_BUFFER_SIZE.

  A third argument, max_buffer_size, is supported, but unused and deprecated.

  BufferedWriter provides or overrides these methods in addition to those from BufferedIOBase and IOBase:

  flush()
  : Force bytes held in the buffer into the raw stream. A BlockingIOError should be raised if the raw stream blocks.

  write(b)
  : Write b, and return the number of bytes written. The object b should be an array of bytes, either bytes, bytearray, or memoryview. When in non-blocking mode, a BlockingIOError is raised if the buffer needs to be written out but the raw stream blocks.

class io.BufferedRandom(raw, buffer_size=DEFAULT_BUFFER_SIZE)
: A buffered interface to random access streams. It inherits BufferedReader and BufferedWriter, and further supports seek() and tell() functionality.

  The constructor creates a reader and writer for a seekable raw stream, given in the first argument. If the buffer_size is omitted it defaults to DEFAULT_BUFFER_SIZE.

  A third argument, max_buffer_size, is supported, but unused and deprecated.

  BufferedRandom is capable of anything BufferedReader or BufferedWriter can do.

class io.BufferedRWPair(reader, writer, buffer_size=DEFAULT_BUFFER_SIZE)
: A buffered I/O object combining two unidirectional RawIOBase objects – one readable, the other writeable – into a single bidirectional endpoint. It inherits BufferedIOBase.

reader and writer are RawIOBase objects that are readable and writeable respectively. If the buffer_size is omitted it defaults to DEFAULT_BUFFER_SIZE.

A fourth argument, max_buffer_size, is supported, but unused and deprecated.

BufferedRWPair implements all of BufferedIOBase's methods except for detach(), which raises UnsupportedOperation.

> Warning: BufferedRWPair does not attempt to synchronize accesses to its underlying raw streams. You should not pass it the same object as reader and writer; use BufferedRandom instead.

15.2.5 Text I/O

class io.TextIOBase
Base class for text streams. This class provides a unicode character and line based interface to stream I/O. There is no readinto() method because Python's unicode strings are immutable. It inherits IOBase. There is no public constructor.

TextIOBase provides or overrides these data attributes and methods in addition to those from IOBase:

encoding
The name of the encoding used to decode the stream's bytes into strings, and to encode strings into bytes.

errors
The error setting of the decoder or encoder.

newlines
A string, a tuple of strings, or None, indicating the newlines translated so far. Depending on the implementation and the initial constructor flags, this may not be available.

buffer
The underlying binary buffer (a BufferedIOBase instance) that TextIOBase deals with. This is not part of the TextIOBase API and may not exist on some implementations.

detach()
Separate the underlying binary buffer from the TextIOBase and return it.

After the underlying buffer has been detached, the TextIOBase is in an unusable state.

Some TextIOBase implementations, like StringIO, may not have the concept of an underlying buffer and calling this method will raise UnsupportedOperation.

New in version 2.7.

read(n=-1)
Read and return at most n characters from the stream as a single unicode. If n is negative or None, reads until EOF.

readline(limit=-1)
Read until newline or EOF and return a single unicode. If the stream is already at EOF, an empty string is returned.

If limit is specified, at most limit characters will be read.

seek(offset[, whence])
Change the stream position to the given offset. Behaviour depends on the whence parameter. The default value for whence is SEEK_SET.

- SEEK_SET or 0: seek from the start of the stream (the default); offset must either be a number returned by TextIOBase.tell(), or zero. Any other offset value produces undefined behaviour.
- SEEK_CUR or 1: "seek" to the current position; offset must be zero, which is a no-operation (all other values are unsupported).
- SEEK_END or 2: seek to the end of the stream; offset must be zero (all other values are unsupported).

Return the new absolute position as an opaque number.

New in version 2.7: The SEEK_* constants.

tell()
Return the current stream position as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.

write(s)
Write the unicode string s to the stream and return the number of characters written.

class io.TextIOWrapper(buffer, encoding=None, errors=None, newline=None, line_buffering=False)
A buffered text stream over a BufferedIOBase binary stream. It inherits TextIOBase.

encoding gives the name of the encoding that the stream will be decoded or encoded with. It defaults to locale.getpreferredencoding().

errors is an optional string that specifies how encoding and decoding errors are to be handled. Pass 'strict' to raise a ValueError exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) 'replace' causes a replacement marker (such as '?') to be inserted where there is malformed data. When writing, 'xmlcharrefreplace' (replace with the appropriate XML character reference) or 'backslashreplace' (replace with backslashed escape sequences) can be used. Any other error handling name that has been registered with codecs.register_error() is also valid.

newline controls how line endings are handled. It can be None, '', '\n', '\r', and '\r\n'. It works as follows:

- On input, if newline is None, universal newlines mode is enabled. Lines in the input can end in '\n', '\r', or '\r\n', and these are translated into '\n' before being returned to the caller. If it is '', universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- On output, if newline is None, any '\n' characters written are translated to the system default line separator, os.linesep. If newline is '', no translation takes place. If newline is any of the other legal values, any '\n' characters written are translated to the given string.

If line_buffering is True, flush() is implied when a call to write contains a newline character or a carriage return.

TextIOWrapper provides one attribute in addition to those of TextIOBase and its parents:

line_buffering
Whether line buffering is enabled.

class io.StringIO(initial_value=u", newline=u'\n')
An in-memory stream for unicode text. It inherits TextIOWrapper.

The initial value of the buffer can be set by providing initial_value. If newline translation is enabled, newlines will be encoded as if by write(). The stream is positioned at the start of the buffer.

The newline argument works like that of TextIOWrapper. The default is to consider only \n characters as ends of lines and to do no newline translation. If newline is set to None, newlines are written as \n on all platforms, but universal newline decoding is still performed when reading.

StringIO provides this method in addition to those from TextIOWrapper and its parents:

getvalue()
: Return a unicode containing the entire contents of the buffer at any time before the StringIO object's close() method is called. Newlines are decoded as if by read(), although the stream position is not changed.

Example usage:

```
import io

output = io.StringIO()
output.write(u'First line.\n')
output.write(u'Second line.\n')

# Retrieve file contents -- this will be
# u'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class io.IncrementalNewlineDecoder
: A helper codec that decodes newlines for universal newlines mode. It inherits codecs.IncrementalDecoder.

15.2.6 Advanced topics

Here we will discuss several advanced topics pertaining to the concrete I/O implementations described above.

Performance

Binary I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O is designed to hide any inefficiency in calling and executing the operating system's unbuffered I/O routines. The gain will vary very much depending on the OS and the kind of I/O which is performed (for example, on some contemporary OSes such as Linux, unbuffered disk I/O can be as fast as buffered I/O). The bottom line, however, is that buffered I/O will offer you predictable performance regardless of the platform and the backing device. Therefore, it is most always preferable to use buffered I/O rather than unbuffered I/O.

Text I/O

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage, because it implies conversions from unicode to binary data using a character codec. This can become noticeable if you handle huge amounts of text data (for example very large log files). Also, TextIOWrapper.tell() and TextIOWrapper.seek() are both quite slow due to the reconstruction algorithm used.

StringIO, however, is a native in-memory unicode container and will exhibit similar speed to BytesIO.

Multi-threading

FileIO objects are thread-safe to the extent that the operating system calls (such as read(2) under Unix) they are wrapping are thread-safe too.

Binary buffered objects (instances of BufferedReader, BufferedWriter, BufferedRandom and BufferedRWPair) protect their internal structures using a lock; it is therefore safe to call them from multiple threads at once.

TextIOWrapper objects are not thread-safe.

Reentrancy

Binary buffered objects (instances of BufferedReader, BufferedWriter, BufferedRandom and BufferedRWPair) are not reentrant. While reentrant calls will not happen in normal situations, they can arise if you are doing I/O in a signal handler. If it is attempted to enter a buffered object again while already being accessed from the same thread, then a RuntimeError is raised.

The above implicitly extends to text files, since the open() function will wrap a buffered object inside a TextIOWrapper. This includes standard streams and therefore affects the built-in function print() as well.

15.3 time — Time access and conversions

This module provides various time-related functions. For related functionality, see also the datetime and calendar modules.

Although this module is always available, not all functions are available on all platforms. Most of the functions defined in this module call platform C library functions with the same name. It may sometimes be helpful to consult the platform documentation, because the semantics of these functions varies among platforms.

An explanation of some terminology and conventions is in order.

- The epoch is the point where the time starts. On January 1st of that year, at 0 hours, the "time since the epoch" is zero. For Unix, the epoch is 1970. To find out what the epoch is, look at gmtime(0).
- The functions in this module do not handle dates and times before the epoch or far in the future. The cut-off point in the future is determined by the C library; for Unix, it is typically in 2038.
- Year 2000 (Y2K) issues: Python depends on the platform's C library, which generally doesn't have year 2000 issues, since all dates and times are represented internally as seconds since the epoch. Functions accepting a struct_time (see below) generally require a 4-digit year. For backward compatibility, 2-digit years are supported if the module variable accept2dyear is a non-zero integer; this variable is initialized to 1 unless the environment variable PYTHONY2K is set to a non-empty string, in which case it is initialized to 0. Thus, you can set PYTHONY2K to a non-empty string in the environment to require 4-digit years for all year input. When 2-digit years are accepted, they are converted according to the POSIX or X/Open standard: values 69-99 are mapped to 1969-1999, and values 0–68 are mapped to 2000–2068. Values 100–1899 are always illegal.
- UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT). The acronym UTC is not a mistake but a compromise between English and French.
- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.

- The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most Unix systems, the clock "ticks" only 50 or 100 times a second.
- On the other hand, the precision of time() and sleep() is better than their Unix equivalents: times are expressed as floating point numbers, time() returns the most accurate time available (using Unix gettimeofday() where available), and sleep() will accept a time with a nonzero fraction (Unix select() is used to implement this, where available).
- The time value as returned by gmtime(), localtime(), and strptime(), and accepted by asctime(), mktime() and strftime(), may be considered as a sequence of 9 integers. The return values of gmtime(), localtime(), and strptime() also offer attribute names for individual fields.

  See struct_time for a description of these objects.

  Changed in version 2.2: The time value sequence was changed from a tuple to a struct_time, with the addition of attribute names for the fields.
- Use the following functions to convert between time representations:

From	To	Use
seconds since the epoch	struct_time in UTC	gmtime()
seconds since the epoch	struct_time in local time	localtime()
struct_time in UTC	seconds since the epoch	calendar.timegm()
struct_time in local time	seconds since the epoch	mktime()

The module defines the following functions and data items:

time.accept2dyear
: Boolean value indicating whether two-digit year values will be accepted. This is true by default, but will be set to false if the environment variable PYTHONY2K has been set to a non-empty string. It may also be modified at run time.

time.altzone
: The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.

time.asctime([t])
: Convert a tuple or struct_time representing a time as returned by gmtime() or localtime() to a 24-character string of the following form: 'Sun Jun 20 23:21:05 1993'. If t is not provided, the current time as returned by localtime() is used. Locale information is not used by asctime().

  Note: Unlike the C function of the same name, there is no trailing newline.

  Changed in version 2.1: Allowed t to be omitted.

time.clock()
: On Unix, return the current processor time as a floating point number expressed in seconds. The precision, and in fact the very definition of the meaning of "processor time", depends on that of the C function of the same name, but in any case, this is the function to use for benchmarking Python or timing algorithms.

  On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function QueryPerformanceCounter(). The resolution is typically better than one microsecond.

time.ctime([secs])
Convert a time expressed in seconds since the epoch to a string representing local time. If secs is not provided or None, the current time as returned by time() is used. ctime(secs) is equivalent to asctime(localtime(secs)). Locale information is not used by ctime().

Changed in version 2.1: Allowed secs to be omitted.

Changed in version 2.4: If secs is None, the current time is used.

time.daylight
Nonzero if a DST timezone is defined.

time.gmtime([secs])
Convert a time expressed in seconds since the epoch to a struct_time in UTC in which the dst flag is always zero. If secs is not provided or None, the current time as returned by time() is used. Fractions of a second are ignored. See above for a description of the struct_time object. See calendar.timegm() for the inverse of this function.

Changed in version 2.1: Allowed secs to be omitted.

Changed in version 2.4: If secs is None, the current time is used.

time.localtime([secs])
Like gmtime() but converts to local time. If secs is not provided or None, the current time as returned by time() is used. The dst flag is set to 1 when DST applies to the given time.

Changed in version 2.1: Allowed secs to be omitted.

Changed in version 2.4: If secs is None, the current time is used.

time.mktime(t)
This is the inverse function of localtime(). Its argument is the struct_time or full 9-tuple (since the dst flag is needed; use -1 as the dst flag if it is unknown) which expresses the time in local time, not UTC. It returns a floating point number, for compatibility with time(). If the input value cannot be represented as a valid time, either OverflowError or ValueError will be raised (which depends on whether the invalid value is caught by Python or the underlying C libraries). The earliest date for which it can generate a time is platform-dependent.

time.sleep(secs)
Suspend execution of the current thread for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time. The actual suspension time may be less than that requested because any caught signal will terminate the sleep() following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system.

time.strftime(format[, t])
Convert a tuple or struct_time representing a time as returned by gmtime() or localtime() to a string as specified by the format argument. If t is not provided, the current time as returned by localtime() is used. format must be a string. ValueError is raised if any field in t is outside of the allowed range. strftime() returns a locale dependent byte string; the result may be converted to unicode by doing strftime(<myformat>).decode(locale.getlocale()[1]).

Changed in version 2.1: Allowed t to be omitted.

Changed in version 2.4: ValueError raised if a field in t is out of range.

Changed in version 2.5: 0 is now a legal argument for any position in the time tuple; if it is normally illegal the value is forced to a correct one.

The following directives can be embedded in the format string. They are shown without the optional field width and precision specification, and are replaced by the indicated characters in the strftime() result:

Directive	Meaning	Notes
%a	Locale's abbreviated weekday name.	
%A	Locale's full weekday name.	
%b	Locale's abbreviated month name.	
%B	Locale's full month name.	
%c	Locale's appropriate date and time representation.	
%d	Day of the month as a decimal number [01,31].	
%H	Hour (24-hour clock) as a decimal number [00,23].	
%I	Hour (12-hour clock) as a decimal number [01,12].	
%j	Day of the year as a decimal number [001,366].	
%m	Month as a decimal number [01,12].	
%M	Minute as a decimal number [00,59].	
%p	Locale's equivalent of either AM or PM.	(1)
%S	Second as a decimal number [00,61].	(2)
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.	(3)
%w	Weekday as a decimal number [0(Sunday),6].	
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.	(3)
%x	Locale's appropriate date representation.	
%X	Locale's appropriate time representation.	
%y	Year without century as a decimal number [00,99].	
%Y	Year with century as a decimal number.	
%Z	Time zone name (no characters if no time zone exists).	
%%	A literal '%' character.	

Notes:

(1) When used with the strptime() function, the %p directive only affects the output hour field if the %I directive is used to parse the hour.

(2) The range really is 0 to 61; this accounts for leap seconds and the (very rare) double leap seconds.

(3) When used with the strptime() function, %U and %W are only used in calculations when the day of the week and the year are specified.

Here is an example, a format for dates compatible with that specified in the RFC 2822 Internet email standard.[1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

Additional directives may be supported on certain platforms, but only the ones listed here have a meaning standardized by ANSI C. To see the full set of format codes supported on your platform, consult the strftime(3) documentation.

[1] The use of %Z is now deprecated, but the %z escape that expands to the preferred hour/minute offset is not supported by all ANSI C libraries. Also, a strict reading of the original 1982 RFC 822 standard calls for a two-digit year (%y rather than %Y), but practice moved to 4-digit years long before the year 2000. After that, RFC 822 became obsolete and the 4-digit year has been first recommended by RFC 1123 and then mandated by RFC 2822.

On some platforms, an optional field width and precision specification can immediately follow the initial '%' of a directive in the following order; this is also not portable. The field width is normally 2 except for %j where it is 3.

time.strptime(string[, format])
Parse a string representing a time according to a format. The return value is a struct_time as returned by gmtime() or localtime().

The format parameter uses the same directives as those used by strftime(); it defaults to "%a %b %d %H:%M:%S %Y" which matches the formatting returned by ctime(). If string cannot be parsed according to format, or if it has excess data after parsing, ValueError is raised. The default values used to fill in any missing data when more accurate values cannot be inferred are (1900, 1, 1, 0, 0, 0, 0, 1, -1).

For example:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")   # doctest: +NORMALIZE_WHITESPACE
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

Support for the %Z directive is based on the values contained in tzname and whether daylight is true. Because of this, it is platform-specific except for recognizing UTC and GMT which are always known (and are considered to be non-daylight savings timezones).

Only the directives specified in the documentation are supported. Because strftime() is implemented per platform it can sometimes offer more directives than those listed. But strptime() is independent of any platform and thus does not necessarily support all directives available that are not documented as supported.

class time.struct_time
The type of the time value sequence returned by gmtime(), localtime(), and strptime(). It is an object with a named tuple interface: values can be accessed by index and by attribute name. The following values are present:

Index	Attribute	Values
0	tm_year	(for example, 1993)
1	tm_mon	range [1, 12]
2	tm_mday	range [1, 31]
3	tm_hour	range [0, 23]
4	tm_min	range [0, 59]
5	tm_sec	range [0, 61]; see (2) in strftime() description
6	tm_wday	range [0, 6], Monday is 0
7	tm_yday	range [1, 366]
8	tm_isdst	0, 1 or -1; see below

New in version 2.2.

Note that unlike the C structure, the month value is a range of [1, 12], not [0, 11]. A year value will be handled as described under Year 2000 (Y2K) issues above.

In calls to mktime(), tm_isdst may be set to 1 when daylight savings time is in effect, and 0 when it is not. A value of -1 indicates that this is not known, and will usually result in the correct state being filled in.

When a tuple with an incorrect length is passed to a function expecting a struct_time, or having elements of the wrong type, a TypeError is raised.

time.time()
Return the time in seconds since the epoch as a floating point number. Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

time.timezone
The offset of the local (non-DST) timezone, in seconds west of UTC (negative in most of Western Europe, positive in the US, zero in the UK).

time.tzname
A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used.

time.tzset()
Reset the time conversion rules used by the library routines. The environment variable TZ specifies how this is done. It will also set the variables tzname (from the TZ environment variable), timezone (non-DST seconds West of UTC), altzone (DST seconds west of UTC) and daylight (to 0 if this timezone does not have any daylight saving time rules, or to nonzero if there is a time, past, present or future when daylight saving time applies).

New in version 2.3.

Availability: Unix.

Note: Although in many cases, changing the TZ environment variable may affect the output of functions like localtime() without calling tzset(), this behavior should not be relied on.

The TZ environment variable should contain no whitespace.

The standard format of the TZ environment variable is (whitespace added for clarity):

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

Where the components are:

std and dst Three or more alphanumerics giving the timezone abbreviations. These will be propagated into time.tzname

offset The offset has the form: ± hh[:mm[:ss]]. This indicates the value added the local time to arrive at UTC. If preceded by a '-', the timezone is east of the Prime Meridian; otherwise, it is west. If no offset follows dst, summer time is assumed to be one hour ahead of standard time.

start[/time], end[/time] Indicates when to change to and back from DST. The format of the start and end dates are one of the following:

Jn The Julian day n (1 <= n <= 365). Leap days are not counted, so in all years February 28 is day 59 and March 1 is day 60.

n The zero-based Julian day (0 <= n <= 365). Leap days are counted, and it is possible to refer to February 29.

Mm.n.d The d'th day (0 <= d <= 6) or week n of month m of the year (1 <= n <= 5, 1 <= m <= 12, where week 5 means "the last d day in month m" which may occur in either the fourth or the fifth week). Week 1 is the first week in which the d'th day occurs. Day zero is Sunday.

time has the same format as offset except that no leading sign ('-' or '+') is allowed. The default, if time is not given, is 02:00:00.

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

On many Unix systems (including *BSD, Linux, Solaris, and Darwin), it is more convenient to use the system's zoneinfo (tzfile(5)) database to specify the timezone rules. To do this, set the TZ environment variable to the path of the required timezone datafile, relative to the root of the systems 'zoneinfo' timezone database, usually located at /usr/share/zoneinfo. For example, 'US/Eastern', 'Australia/Melbourne', 'Egypt' or 'Europe/Amsterdam'.

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

See also:

Module datetime More object-oriented interface to dates and times.

Module locale Internationalization services. The locale setting affects the interpretation of many format specifiers in strftime() and strptime().

Module calendar General calendar-related functions. timegm() is the inverse of gmtime() from this module.

15.4 argparse — Parser for command-line options, arguments and sub-commands

New in version 2.7.

Source code: Lib/argparse.py

Tutorial

This page contains the API reference information. For a more gentle introduction to Python command-line parsing, have a look at the argparse tutorial.

The argparse module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and argparse will figure out how to parse those out of sys.argv. The argparse module also automatically generates help and usage messages and issues errors when users give the program invalid arguments.

15.4.1 Example

The following code is a Python program that takes a list of integers and produces either the sum or the max:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print args.accumulate(args.integers)
```

Assuming the Python code above is saved into a file called prog.py, it can be run at the command line and provides useful help messages:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

When run with the appropriate arguments, it prints either the sum or the max of the command-line integers:

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

If invalid arguments are passed in, it will issue an error:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

The following sections walk you through this example.

Creating a parser

The first step in using the argparse is creating an ArgumentParser object:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

The ArgumentParser object will hold all the information necessary to parse the command line into Python data types.

Adding arguments

Filling an ArgumentParser with information about program arguments is done by making calls to the add_argument() method. Generally, these calls tell the ArgumentParser how to take the strings on the command line and turn them into objects. This information is stored and used when parse_args() is called. For example:

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
...                     const=sum, default=max,
...                     help='sum the integers (default: find the max)')
```

Later, calling parse_args() will return an object with two attributes, integers and accumulate. The integers attribute will be a list of one or more ints, and the accumulate attribute will be either the sum() function, if --sum was specified at the command line, or the max() function if it was not.

Parsing arguments

ArgumentParser parses arguments through the parse_args() method. This will inspect the command line, convert each argument to the appropriate type and then invoke the appropriate action. In most cases, this means a simple Namespace object will be built up from attributes parsed out of the command line:

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

In a script, parse_args() will typically be called with no arguments, and the ArgumentParser will automatically determine the command-line arguments from sys.argv.

15.4.2 ArgumentParser objects

class argparse.ArgumentParser(prog=None, usage=None, description=None, epilog=None, parents=[], formatter_class=argparse.HelpFormatter, prefix_chars='-', fromfile_prefix_chars=None, argument_default=None, conflict_handler='error', add_help=True)

Create a new ArgumentParser object. All parameters should be passed as keyword arguments. Each parameter has its own more detailed description below, but in short they are:

- prog - The name of the program (default: sys.argv[0])
- usage - The string describing the program usage (default: generated from arguments added to parser)
- description - Text to display before the argument help (default: none)
- epilog - Text to display after the argument help (default: none)
- parents - A list of ArgumentParser objects whose arguments should also be included
- formatter_class - A class for customizing the help output
- prefix_chars - The set of characters that prefix optional arguments (default: '-')
- fromfile_prefix_chars - The set of characters that prefix files from which additional arguments should be read (default: None)
- argument_default - The global default value for arguments (default: None)

- conflict_handler - The strategy for resolving conflicting optionals (usually unnecessary)
- add_help - Add a -h/--help option to the parser (default: True)

The following sections describe how each of these are used.

prog

By default, ArgumentParser objects use sys.argv[0] to determine how to display the name of the program in help messages. This default is almost always desirable because it will make the help messages match how the program was invoked on the command line. For example, consider a file named myprogram.py with the following code:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

The help for this program will display myprogram.py as the program name (regardless of where the program was invoked from):

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
$ cd ..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

To change this default behavior, another value can be supplied using the prog= argument to ArgumentParser:

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
 -h, --help  show this help message and exit
```

Note that the program name, whether determined from sys.argv[0] or from the prog= argument, is available to help messages using the %(prog)s format specifier.

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

usage

By default, ArgumentParser calculates the usage message from the arguments it contains:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

The default message can be overridden with the usage= keyword argument:

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

The %(prog)s format specifier is available to fill in the program name in your usage messages.

description

Most calls to the ArgumentParser constructor will use the description= keyword argument. This argument gives a brief description of what the program does and how it works. In help messages, the description is displayed between the command-line usage string and the help messages for the various arguments:

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit
```

By default, the description will be line-wrapped so that it fits within the given space. To change this behavior, see the formatter_class argument.

epilog

Some programs like to display additional description of the program after the description of the arguments. Such text can be specified using the epilog= argument to ArgumentParser:

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

As with the description argument, the epilog= text is by default line-wrapped, but this behavior can be adjusted with the formatter_class argument to ArgumentParser.

parents

Sometimes, several parsers share a common set of arguments. Rather than repeating the definitions of these arguments, a single parser with all the shared arguments and passed to parents= argument to ArgumentParser can be used. The parents= argument takes a list of ArgumentParser objects, collects all the positional and optional actions from them, and adds these actions to the ArgumentParser object being constructed:

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

Note that most parent parsers will specify add_help=False. Otherwise, the ArgumentParser will see two -h/--help options (one in the parent and one in the child) and raise an error.

Note: You must fully initialize the parsers before passing them via parents=. If you change the parent parsers after the child parser, those changes will not be reflected in the child.

---

formatter_class

ArgumentParser objects allow the help formatting to be customized by specifying an alternate formatting class. Currently, there are three such classes:

class argparse.RawDescriptionHelpFormatter
class argparse.RawTextHelpFormatter
class argparse.ArgumentDefaultsHelpFormatter

The first two allow more control over how textual descriptions are displayed, while the last automatically adds information about argument default values.

By default, ArgumentParser objects line-wrap the description and epilog texts in command-line help messages:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

optional arguments:
 -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

Passing RawDescriptionHelpFormatter as formatter_class= indicates that description and epilog are already correctly formatted and should not be line-wrapped:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

optional arguments:
 -h, --help  show this help message and exit
```

RawTextHelpFormatter maintains whitespace for all sorts of help text, including argument descriptions. However, multiple new lines are replaced with one. If you wish to preserve multiple blank lines, add spaces between the newlines.

The other formatter class available, ArgumentDefaultsHelpFormatter, will add information about the default value of each of the arguments:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
```

(continues on next page)

(continued from previous page)

```
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

prefix_chars

Most command-line options will use - as the prefix, e.g. -f/--foo. Parsers that need to support different or additional prefix characters, e.g. for options like +f or /foo, may specify them using the prefix_chars= argument to the ArgumentParser constructor:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

The prefix_chars= argument defaults to `'-'`. Supplying a set of characters that does not include - will cause -f/--foo options to be disallowed.

fromfile_prefix_chars

Sometimes, for example when dealing with a particularly long argument lists, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the fromfile_prefix_chars= argument is given to the ArgumentParser constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example:

```
>>> with open('args.txt', 'w') as fp:
...     fp.write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

Arguments read from a file must by default be one per line (but see also convert_arg_line_to_args()) and are treated as if they were in the same place as the original file referencing argument on the command line. So in the example above, the expression `['-f', 'foo', '@args.txt']` is considered equivalent to the expression `['-f', 'foo', '-f', 'bar']`.

The fromfile_prefix_chars= argument defaults to None, meaning that arguments will never be treated as file references.

argument_default

Generally, argument defaults are specified either by passing a default to add_argument() or by calling the set_defaults() methods with a specific set of name-value pairs. Sometimes however, it may be useful to specify

a single parser-wide default for arguments. This can be accomplished by passing the argument_default= keyword argument to ArgumentParser. For example, to globally suppress attribute creation on parse_args() calls, we supply argument_default=SUPPRESS:

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

conflict_handler

ArgumentParser objects do not allow two actions with the same option string. By default, ArgumentParser objects raise an exception if an attempt is made to create an argument with an option string that is already in use:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

Sometimes (e.g. when using parents) it may be useful to simply override any older arguments with the same option string. To get this behavior, the value 'resolve' can be supplied to the conflict_handler= argument of ArgumentParser:

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

Note that ArgumentParser objects only remove an action if all of its option strings are overridden. So, in the example above, the old -f/--foo action is retained as the -f action, because only the --foo option string was overridden.

add_help

By default, ArgumentParser objects add an option which simply displays the parser's help message. For example, consider a file named myprogram.py containing the following code:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

If -h or --help is supplied at the command line, the ArgumentParser help will be printed:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

Occasionally, it may be useful to disable the addition of this help option. This can be achieved by passing False as the add_help= argument to ArgumentParser:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

optional arguments:
 --foo FOO  foo help
```

The help option is typically -h/--help. The exception to this is if the prefix_chars= is specified and does not include -, in which case -h and --help are not valid options. In this case, the first character in prefix_chars is used to prefix the help options:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

optional arguments:
  +h, ++help  show this help message and exit
```

15.4.3 The add_argument() method

ArgumentParser.add_argument(name or flags... [, action] [, nargs] [, const] [, default] [, type] [, choices] [, required] [, help] [, metavar] [, dest])

Define how a single command-line argument should be parsed. Each parameter has its own more detailed description below, but in short they are:

- name or flags - Either a name or a list of option strings, e.g. foo or -f, --foo.
- action - The basic type of action to be taken when this argument is encountered at the command line.
- nargs - The number of command-line arguments that should be consumed.
- const - A constant value required by some action and nargs selections.
- default - The value produced if the argument is absent from the command line.
- type - The type to which the command-line argument should be converted.
- choices - A container of the allowable values for the argument.
- required - Whether or not the command-line option may be omitted (optionals only).
- help - A brief description of what the argument does.
- metavar - A name for the argument in usage messages.
- dest - The name of the attribute to be added to the object returned by parse_args().

The following sections describe how each of these are used.

name or flags

The add_argument() method must know whether an optional argument, like -f or --foo, or a positional argument, like a list of filenames, is expected. The first arguments passed to add_argument() must therefore be either a series of flags, or a simple argument name. For example, an optional argument could be created like:

```
>>> parser.add_argument('-f', '--foo')
```

while a positional argument could be created like:

```
>>> parser.add_argument('bar')
```

When parse_args() is called, optional arguments will be identified by the - prefix, and the remaining arguments will be assumed to be positional:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: too few arguments
```

action

ArgumentParser objects associate command-line arguments with actions. These actions can do just about anything with the command-line arguments associated with them, though most actions simply add an attribute to the object returned by parse_args(). The action keyword argument specifies how the command-line arguments should be handled. The supplied actions are:

- 'store' - This just stores the argument's value. This is the default action. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args('--foo 1'.split())
Namespace(foo='1')
```

- 'store_const' - This stores the value specified by the const keyword argument. The 'store_const' action is most commonly used with optional arguments that specify some sort of flag. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const', const=42)
>>> parser.parse_args(['--foo'])
Namespace(foo=42)
```

- 'store_true' and 'store_false' - These are special cases of 'store_const' using for storing the values True and False respectively. In addition, they create default values of False and True respectively. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
```

(continues on next page)

(continued from previous page)

```
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(bar=False, baz=True, foo=True)
```

- `'append'` - This stores a list, and appends each argument value to the list. This is useful to allow an option to be specified multiple times. Example usage:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- `'append_const'` - This stores a list, and appends the value specified by the const keyword argument to the list. (Note that the const keyword argument defaults to None.) The `'append_const'` action is typically useful when multiple arguments need to store constants to the same list. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<type 'str'>, <type 'int'>])
```

- `'count'` - This counts the number of times a keyword argument occurs. For example, this is useful for increasing verbosity levels:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count')
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

- `'help'` - This prints a complete help message for all the options in the current parser and then exits. By default a help action is automatically added to the parser. See ArgumentParser for details of how the output is created.

- `'version'` - This expects a version= keyword argument in the add_argument() call, and prints version information and exits when invoked:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

You may also specify an arbitrary action by passing an Action subclass or other object that implements the same interface. The recommended way to do this is to extend Action, overriding the `__call__` method and optionally the `__init__` method.

An example of a custom action:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super(FooAction, self).__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
```

(continues on next page)

(continued from previous page)

```
...         print '%r %r %r' % (namespace, values, option_string)
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

For more details, see Action.

nargs

ArgumentParser objects usually associate a single command-line argument with a single action to be taken. The nargs keyword argument associates a different number of command-line arguments with a single action. The supported values are:

- N (an integer). N arguments from the command line will be gathered together into a list. For example:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs=2)
  >>> parser.add_argument('bar', nargs=1)
  >>> parser.parse_args('c --foo a b'.split())
  Namespace(bar=['c'], foo=['a', 'b'])
  ```

  Note that nargs=1 produces a list of one item. This is different from the default, in which the item is produced by itself.

- `'?'`. One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument is present, the value from default will be produced. Note that for optional arguments, there is an additional case - the option string is present but not followed by a command-line argument. In this case the value from const will be produced. Some examples to illustrate this:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs='?', const='c', default='d')
  >>> parser.add_argument('bar', nargs='?', default='d')
  >>> parser.parse_args(['XX', '--foo', 'YY'])
  Namespace(bar='XX', foo='YY')
  >>> parser.parse_args(['XX', '--foo'])
  Namespace(bar='XX', foo='c')
  >>> parser.parse_args([])
  Namespace(bar='d', foo='d')
  ```

  One of the more common uses of nargs=`'?'` is to allow optional input and output files:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
  ...                     default=sys.stdin)
  >>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
  ...                     default=sys.stdout)
  >>> parser.parse_args(['input.txt', 'output.txt'])
  ```

(continues on next page)

(continued from previous page)

```
Namespace(infile=<open file 'input.txt', mode 'r' at 0x...>,
          outfile=<open file 'output.txt', mode 'w' at 0x...>)
>>> parser.parse_args([])
Namespace(infile=<open file '<stdin>', mode 'r' at 0x...>,
          outfile=<open file '<stdout>', mode 'w' at 0x...>)
```

- `'*'`. All command-line arguments present are gathered into a list. Note that it generally doesn't make much sense to have more than one positional argument with nargs=`'*'`, but multiple optional arguments with nargs=`'*'` is possible. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- `'+'`. Just like `'*'`, all command-line args present are gathered into a list. Additionally, an error message will be generated if there wasn't at least one command-line argument present. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: too few arguments
```

- argparse.REMAINDER. All the remaining command-line arguments are gathered into a list. This is commonly useful for command line utilities that dispatch to other command line utilities:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print parser.parse_args('--foo B cmd --arg1 XX ZZ'.split())
Namespace(args=['--arg1', 'XX', 'ZZ'], command='cmd', foo='B')
```

If the nargs keyword argument is not provided, the number of arguments consumed is determined by the action. Generally this means a single command-line argument will be consumed and a single item (not a list) will be produced.

const

The const argument of add_argument() is used to hold constant values that are not read from the command line but are required for the various ArgumentParser actions. The two most common uses of it are:

- When add_argument() is called with action=`'store_const'` or action=`'append_const'`. These actions add the const value to one of the attributes of the object returned by parse_args(). See the action description for examples.
- When add_argument() is called with option strings (like -f or --foo) and nargs=`'?'`. This creates an optional argument that can be followed by zero or one command-line arguments. When parsing the command line, if the option string is encountered with no command-line argument following it, the value of const will be assumed instead. See the nargs description for examples.

With the 'store_const' and 'append_const' actions, the const keyword argument must be given. For other actions, it defaults to None.

default

All optional arguments and some positional arguments may be omitted at the command line. The default keyword argument of add_argument(), whose value defaults to None, specifies what value should be used if the command-line argument is not present. For optional arguments, the default value is used when the option string was not present at the command line:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

If the default value is a string, the parser parses the value as if it were a command-line argument. In particular, the parser applies any type conversion argument, if provided, before setting the attribute on the Namespace return value. Otherwise, the parser uses the value as is:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

For positional arguments with nargs equal to ? or *, the default value is used when no command-line argument was present:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

Providing default=argparse.SUPPRESS causes no attribute to be added if the command-line argument was not present.:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type

By default, ArgumentParser objects read command-line arguments in as simple strings. However, quite often the command-line string should instead be interpreted as another type, like a float or int. The type keyword argument of add_argument() allows any necessary type-checking and type conversions to be performed. Common built-in types and functions can be used directly as the value of the type argument:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=file)
>>> parser.parse_args('2 temp.txt'.split())
Namespace(bar=<open file 'temp.txt', mode 'r' at 0x...>, foo=2)
```

See the section on the default keyword argument for information on when the type argument is applied to default arguments.

To ease the use of various types of files, the argparse module provides the factory FileType which takes the mode= and bufsize= arguments of the file object. For example, FileType('w') can be used to create a writable file:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<open file 'out.txt', mode 'w' at 0x...>)
```

type= can take any callable that takes a single string argument and returns the converted value:

```
>>> def perfect_square(string):
...     value = int(string)
...     sqrt = math.sqrt(value)
...     if sqrt != int(sqrt):
...         msg = "%r is not a perfect square" % string
...         raise argparse.ArgumentTypeError(msg)
...     return value
...
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
>>> parser.parse_args(['9'])
Namespace(foo=9)
>>> parser.parse_args(['7'])
usage: PROG [-h] foo
PROG: error: argument foo: '7' is not a perfect square
```

The choices keyword argument may be more convenient for type checkers that simply check against a range of values:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=xrange(5, 10))
>>> parser.parse_args(['7'])
Namespace(foo=7)
>>> parser.parse_args(['11'])
usage: PROG [-h] {5,6,7,8,9}
PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)
```

See the choices section for more details.

choices

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a container object as the choices keyword argument to add_argument(). When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the acceptable values:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

Note that inclusion in the choices container is checked after any type conversions have been performed, so the type of the objects in the choices container should match the type specified:

```
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

Any object that supports the in operator can be passed as the choices value, so dict objects, set objects, custom containers, etc. are all supported.

required

In general, the argparse module assumes that flags like -f and --bar indicate optional arguments, which can always be omitted at the command line. To make an option required, True can be specified for the required= keyword argument to add_argument():

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: argparse.py [-h] [--foo FOO]
argparse.py: error: option --foo is required
```

As the example shows, if an option is marked as required, parse_args() will report an error if that option is not present at the command line.

Note: Required options are generally considered bad form because users expect options to be optional, and thus they should be avoided when possible.

---

help

The help value is a string containing a brief description of the argument. When a user requests help (usually by using -h or --help at the command line), these help descriptions will be displayed with each argument:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
```

(continues on next page)

(continued from previous page)

```
...                     help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
 bar     one of the bars to be frobbled

optional arguments:
 -h, --help  show this help message and exit
 --foo   foo the bars before frobbling
```

The help strings can include various format specifiers to avoid repetition of things like the program name or the argument default. The available specifiers include the program name, %(prog)s and most keyword arguments to add_argument(), e.g. %(default)s, %(type)s, etc.:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

optional arguments:
 -h, --help  show this help message and exit
```

argparse supports silencing the help entry for certain options, by setting the help value to argparse.SUPPRESS:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

optional arguments:
  -h, --help  show this help message and exit
```

metavar

When ArgumentParser generates help messages, it needs some way to refer to each expected argument. By default, ArgumentParser objects use the dest value as the "name" of each object. By default, for positional argument actions, the dest value is used directly, and for optional argument actions, the dest value is uppercased. So, a single positional argument with dest='bar' will be referred to as bar. A single optional argument --foo that should be followed by a single command-line argument will be referred to as FOO. An example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar
```

(continues on next page)

(continued from previous page)

```
positional arguments:
 bar

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO
```

An alternative name can be specified with metavar:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

optional arguments:
 -h, --help  show this help message and exit
 --foo YYY
```

Note that metavar only changes the displayed name - the name of the attribute on the parse_args() object is still determined by the dest value.

Different values of nargs may cause the metavar to be used multiple times. Providing a tuple to metavar specifies a different display for each of the arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:
 -h, --help     show this help message and exit
 -x X X
 --foo bar baz
```

dest

Most ArgumentParser actions add some value as an attribute of the object returned by parse_args(). The name of this attribute is determined by the dest keyword argument of add_argument(). For positional argument actions, dest is normally supplied as the first argument to add_argument():

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

For optional argument actions, the value of dest is normally inferred from the option strings. ArgumentParser generates the value of dest by taking the first long option string and stripping away the initial -- string. If

no long option strings were supplied, dest will be derived from the first short option string by stripping the initial - character. Any internal - characters will be converted to _ characters to make sure the string is a valid attribute name. The examples below illustrate this behavior:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

dest allows a custom attribute name to be provided:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

Action classes

Action classes implement the Action API, a callable which returns a callable which processes arguments from the command-line. Any object which follows this API may be passed as the action parameter to add_argument().

class argparse.Action(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)

Action objects are used by an ArgumentParser to represent the information needed to parse a single argument from one or more strings from the command line. The Action class must accept the two positional arguments plus any keyword arguments passed to ArgumentParser.add_argument() except for the action itself.

Instances of Action (or return value of any callable to the action parameter) should have attributes "dest", "option_strings", "default", "type", "required", "help", etc. defined. The easiest way to ensure these attributes are defined is to call Action.__init__.

Action instances should be callable, so subclasses must override the __call__ method, which should accept four parameters:

- parser - The ArgumentParser object which contains this action.
- namespace - The Namespace object that will be returned by parse_args(). Most actions add an attribute to this object using setattr().
- values - The associated command-line arguments, with any type conversions applied. Type conversions are specified with the type keyword argument to add_argument().
- option_string - The option string that was used to invoke this action. The option_string argument is optional, and will be absent if the action is associated with a positional argument.

The __call__ method may perform arbitrary actions, but will typically set attributes on the namespace based on dest and values.

15.4.4 The parse_args() method

ArgumentParser.parse_args(args=None, namespace=None)

Convert argument strings to objects and assign them as attributes of the namespace. Return the populated namespace.

Previous calls to add_argument() determine exactly what objects are created and how they are assigned. See the documentation for add_argument() for details.

- args - List of strings to parse. The default is taken from sys.argv.
- namespace - An object to take the attributes. The default is a new empty Namespace object.

Option value syntax

The parse_args() method supports several ways of specifying the value of an option (if it takes one). In the simplest case, the option and its value are passed as two separate arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

For long options (options with names longer than a single character), the option and value can also be passed as a single command-line argument, using = to separate them:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

For short options (options only one character long), the option and its value can be concatenated:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

Several short options can be joined together, using only a single - prefix, as long as only the last option (or none of them) requires a value:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

Invalid arguments

While parsing the command line, parse_args() checks for a variety of errors, including ambiguous options, invalid types, invalid options, wrong number of positional arguments, etc. When it encounters such an error, it exits and prints the error along with a usage message:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'
```

(continues on next page)

(continued from previous page)

```
>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

Arguments containing -

The parse_args() method attempts to give errors whenever the user has clearly made a mistake, but some situations are inherently ambiguous. For example, the command-line argument -1 could either be an attempt to specify an option or an attempt to provide a positional argument. The parse_args() method is cautious here: positional arguments may only begin with - if they look like negative numbers and there are no options in the parser that look like negative numbers:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

If you have positional arguments that must begin with - and don't look like negative numbers, you can insert the pseudo-argument '--' which tells parse_args() that everything after that is a positional argument:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

Argument abbreviations (prefix matching)

The parse_args() method allows long options to be abbreviated to a prefix, if the abbreviation is unambiguous (the prefix matches a unique option):

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

An error is produced for arguments that could produce more than one options.

Beyond sys.argv

Sometimes it may be useful to have an ArgumentParser parse arguments other than those of sys.argv. This can be accomplished by passing a list of strings to parse_args(). This is useful for testing at the interactive prompt:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=xrange(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

The Namespace object

class argparse.Namespace
Simple class used by default by parse_args() to create an object holding attributes and return it.

This class is deliberately simple, just an object subclass with a readable string representation. If you prefer to have dict-like view of the attributes, you can use the standard Python idiom, vars():

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

It may also be useful to have an ArgumentParser assign attributes to an already existing object, rather than a new Namespace object. This can be achieved by specifying the namespace= keyword argument:

```
>>> class C(object):
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

15.4.5 Other utilities

Sub-commands

ArgumentParser.add_subparsers([title][, description][, prog][, parser_class][, action][, option_string][, dest][, help][, metavar])

Many programs split up their functionality into a number of sub-commands, for example, the svn program can invoke sub-commands like svn checkout, svn update, and svn commit. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. ArgumentParser supports the creation of such sub-commands with the add_subparsers() method. The add_subparsers() method is normally called with no arguments and returns a special action object. This object has a single method, add_parser(), which takes a command name and any ArgumentParser constructor arguments, and returns an ArgumentParser object that can be modified as usual.

Description of parameters:

- title - title for the sub-parser group in help output; by default "subcommands" if description is provided, otherwise uses title for positional arguments
- description - description for the sub-parser group in help output, by default None
- prog - usage information that will be displayed with sub-command help, by default the name of the program and any positional arguments before the subparser argument
- parser_class - class which will be used to create sub-parser instances, by default the class of the current parser (e.g. ArgumentParser)
- action - the basic type of action to be taken when this argument is encountered at the command line
- dest - name of the attribute under which sub-command name will be stored; by default None and no value is stored
- help - help for sub-parser group in help output, by default None
- metavar - string presenting available sub-commands in help; by default it is None and presents sub-commands in form {cmd1, cmd2, ..}

Some example usage:

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>>
>>> # create the parser for the "a" command
```

(continues on next page)

(continued from previous page)

```
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

Note that the object returned by parse_args() will only contain attributes for the main parser and the subparser that was selected by the command line (and not any other subparsers). So in the example above, when the a command is specified, only the foo and bar attributes are present, and when the b command is specified, only the foo and baz attributes are present.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (A help message for each subparser command, however, can be given by supplying the help= argument to add_parser() as above.)

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   sub-command help
    a     a help
    b     b help

optional arguments:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

optional arguments:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

The add_subparsers() method also supports title and description keyword arguments. When either is present, the subparser's commands will appear in their own group in the help output. For example:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
```

(continues on next page)

(continued from previous page)

```
...                                 description='valid subcommands',
...                                 help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

One particularly effective way of handling sub-commands is to combine the use of the add_subparsers() method with calls to set_defaults() so that each subparser knows which Python function it should execute. For example:

```
>>> # sub-command functions
>>> def foo(args):
...     print args.x * args.y
...
>>> def bar(args):
...     print '((%s))' % args.z
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

This way, you can let parse_args() do the job of calling the appropriate function after argument parsing is complete. Associating functions with actions like this is typically the easiest way to handle the different actions for each of your subparsers. However, if it is necessary to check the name of the subparser that was invoked, the dest keyword argument to the add_subparsers() call will work:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

FileType objects

class argparse.FileType(mode='r', bufsize=None)

The FileType factory creates objects that can be passed to the type argument of ArgumentParser.add_argument(). Arguments that have FileType objects as their type will open command-line arguments as files with the requested modes and buffer sizes:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--output', type=argparse.FileType('wb', 0))
>>> parser.parse_args(['--output', 'out'])
Namespace(output=<open file 'out', mode 'wb' at 0x...>)
```

FileType objects understand the pseudo-argument `'-'` and automatically convert this into sys.stdin for readable FileType objects and sys.stdout for writable FileType objects:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<open file '<stdin>', mode 'r' at 0x...>)
```

Argument groups

ArgumentParser.add_argument_group(title=None, description=None)

By default, ArgumentParser groups command-line arguments into "positional arguments" and "optional arguments" when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the add_argument_group() method:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

The add_argument_group() method returns an argument group object which has an add_argument() method just like a regular ArgumentParser. When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The add_argument_group() method accepts title and description arguments which can be used to customize this display:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

Note that any arguments not in your user-defined groups will end up back in the usual "positional arguments" and "optional arguments" sections.

Mutual exclusion

ArgumentParser.add_mutually_exclusive_group(required=False)

Create a mutually exclusive group. argparse will make sure that only one of the arguments in the mutually exclusive group was present on the command line:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

The add_mutually_exclusive_group() method also accepts a required argument, to indicate that at least one of the mutually exclusive arguments is required:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

Note that currently mutually exclusive argument groups do not support the title and description arguments of add_argument_group().

Parser defaults

ArgumentParser.set_defaults(**kwargs)
Most of the time, the attributes of the object returned by parse_args() will be fully determined by inspecting the command-line arguments and the argument actions. set_defaults() allows some additional attributes that are determined without any inspection of the command line to be added:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

Note that parser-level defaults always override argument-level defaults:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

Parser-level defaults can be particularly useful when working with multiple parsers. See the add_subparsers() method for an example of this type.

ArgumentParser.get_default(dest)
Get the default value for a namespace attribute, as set by either add_argument() or by set_defaults():

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

Printing help

In most typical applications, parse_args() will take care of formatting and printing any usage or error messages. However, several formatting methods are available:

ArgumentParser.print_usage(file=None)
Print a brief description of how the ArgumentParser should be invoked on the command line. If file is None, sys.stdout is assumed.

ArgumentParser.print_help(file=None)
Print a help message, including the program usage and information about the arguments registered with the ArgumentParser. If file is None, sys.stdout is assumed.

There are also variants of these methods that simply return a string instead of printing it:

ArgumentParser.format_usage()
Return a string containing a brief description of how the ArgumentParser should be invoked on the command line.

ArgumentParser.format_help()
Return a string containing a help message, including the program usage and information about the arguments registered with the ArgumentParser.

Partial parsing

ArgumentParser.parse_known_args(args=None, namespace=None)

Sometimes a script may only parse a few of the command-line arguments, passing the remaining arguments on to another script or program. In these cases, the parse_known_args() method can be useful. It works much like parse_args() except that it does not produce an error when extra arguments are present. Instead, it returns a two item tuple containing the populated namespace and the list of remaining argument strings.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

> **Warning:** Prefix matching rules apply to parse_known_args(). The parser may consume an option even if it's just a prefix of one of its known options, instead of leaving it in the remaining arguments list.

Customizing file parsing

ArgumentParser.convert_arg_line_to_args(arg_line)

Arguments that are read from a file (see the fromfile_prefix_chars keyword argument to the ArgumentParser constructor) are read one argument per line. convert_arg_line_to_args() can be overridden for fancier reading.

This method takes a single argument arg_line which is a string read from the argument file. It returns a list of arguments parsed from this string. The method is called once per line read from the argument file, in order.

A useful override of this method is one that treats each space-separated word as an argument:

```
def convert_arg_line_to_args(self, arg_line):
    return arg_line.split()
```

Exiting methods

ArgumentParser.exit(status=0, message=None)

This method terminates the program, exiting with the specified status and, if given, it prints a message before that.

ArgumentParser.error(message)

This method prints a usage message including the message to the standard error and terminates the program with a status code of 2.

15.4.6 Upgrading optparse code

Originally, the argparse module had attempted to maintain compatibility with optparse. However, optparse was difficult to extend transparently, particularly with the changes required to support the new nargs= specifiers and better usage messages. When most everything in optparse had either been copy-pasted over or monkey-patched, it no longer seemed practical to try to maintain the backwards compatibility.

The argparse module improves on the standard library optparse module in a number of ways including:

- Handling positional arguments.
- Supporting sub-commands.
- Allowing alternative option prefixes like + and /.
- Handling zero-or-more and one-or-more style arguments.
- Producing more informative usage messages.
- Providing a much simpler interface for custom type and action.

A partial upgrade path from optparse to argparse:

- Replace all optparse.OptionParser.add_option() calls with ArgumentParser.add_argument() calls.
- Replace (options, args) = parser.parse_args() with args = parser.parse_args() and add additional ArgumentParser.add_argument() calls for the positional arguments. Keep in mind that what was previously called options, now in the argparse context is called args.
- Replace optparse.OptionParser.disable_interspersed_args() by setting nargs of a positional argument to argparse.REMAINDER, or use parse_known_args() to collect unparsed argument strings in a separate list.
- Replace callback actions and the callback_* keyword arguments with type or action arguments.
- Replace string names for type keyword arguments with the corresponding type objects (e.g. int, float, complex, etc).
- Replace optparse.Values with Namespace and optparse.OptionError and optparse.OptionValueError with ArgumentError.
- Replace strings with implicit arguments such as %default or %prog with the standard Python syntax to use dictionaries to format strings, that is, %(default)s and %(prog)s.
- Replace the OptionParser constructor version argument with a call to parser.add_argument('--version', action='version', version='<the version>').

15.5 optparse — Parser for command line options

New in version 2.3.

Deprecated since version 2.7: The optparse module is deprecated and will not be developed further; development will continue with the argparse module.

Source code: Lib/optparse.py

---

optparse is a more convenient, flexible, and powerful library for parsing command-line options than the old getopt module. optparse uses a more declarative style of command-line parsing: you create an instance of OptionParser, populate it with options, and parse the command line. optparse allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here's an example of using optparse in a simple script:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
```

(continues on next page)

(continued from previous page)

```
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

With these few lines of code, users of your script can now do the "usual thing" on the command-line, for example:

```
<yourscript> --file=outfile -q
```

As it parses the command line, optparse sets attributes of the options object returned by parse_args() based on user-supplied command-line values. When parse_args() returns from parsing this command line, options.filename will be "outfile" and options.verbose will be False. optparse supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of

```
<yourscript> -h
<yourscript> --help
```

and optparse will print out a brief summary of your script's options:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

where the value of yourscript is determined at runtime (normally from sys.argv[0]).

15.5.1 Background

optparse was explicitly designed to encourage the creation of programs with straightforward, conventional command-line interfaces. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, read this section to acquaint yourself with them.

Terminology

argument a string entered on the command-line, and passed by the shell to execl() or execv(). In Python, arguments are elements of sys.argv[1:] (sys.argv[0] is the name of the program being executed). Unix shells also use the term "word".

It is occasionally desirable to substitute an argument list other than sys.argv[1:], so you should read "argument" as "an element of sys.argv[1:], or of some other list provided as a substitute for sys.argv[1:]".

option an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen ("-") followed by a single letter, e.g. -x or -F. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. -x -F is equivalent to -xF. The GNU project introduced -- followed by a series of hyphen-separated words, e.g. --file or --dry-run. These are the only two option syntaxes provided by optparse.

Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. -pf (this is not the same as multiple options merged into a single argument)
- a hyphen followed by a whole word, e.g. -file (this is technically equivalent to the previous syntax, but they aren't usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. +f, +rgb
- a slash followed by a letter, or a few letters, or a word, e.g. /f, /file

These option syntaxes are not supported by optparse, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting VMS, MS-DOS, and/or Windows.

option argument an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With optparse, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an "optional option arguments" feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous: if -a takes an optional argument and -b is another option entirely, how do we interpret -ab? Because of this ambiguity, optparse does not support this feature.

positional argument something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

required option an option that must be supplied on the command-line; note that the phrase "required option" is self-contradictory in English. optparse doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line:

```
prog -v --report report.txt foo bar
```

-v and --report are both options. Assuming that --report takes one argument, report.txt is an option argument. foo and bar are positional arguments.

What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually optional. A program should be able to run just fine with no options

whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are find, tar, and dd—all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have "required options". Think about it. If it's required, then it's not optional! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble cp utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, cp fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most cp implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of cp, which is to copy either one file to another, or several files to another directory.

What are positional arguments for?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter how you get that information from the user—most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply—use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the "Preferences" dialog of a GUI, or command-line options—the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.

15.5.2 Tutorial

While optparse is quite flexible and powerful, it's also straightforward to use in most cases. This section covers the code patterns that are common to any optparse-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Each option has one or more option strings, such as -f or --file, and several option attributes that tell optparse what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```
parser.add_option("-f", "--file", ...)
```

You're free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to OptionParser.add_option() are effectively labels for the option defined by that call. For brevity, we will frequently refer to encountering an option on the command line; in reality, optparse encounters option strings and looks up options from them.

Once all of your options are defined, instruct optparse to parse your program's command line:

```
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to parse_args(), but that's rarely necessary: by default it uses sys.argv[1:].)

parse_args() returns two values:

- options, an object containing values for all of your options—e.g. if --file takes a single string argument, then options.file will be the filename supplied by the user, or None if the user did not supply that option
- args, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: action, type, dest (destination), and help. Of these, action is the most fundamental.

Understanding option actions

Actions tell optparse what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into optparse; adding new actions is an advanced topic covered in section Extending optparse. Most actions tell optparse to store a value in some variable—for example, take a string from the command line and store it in an attribute of options.

If you don't specify an option action, optparse defaults to store.

The store action

The most common option action is store, which tells optparse to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

For example:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

Now let's make up a fake command line and ask optparse to parse it:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When optparse sees the option string -f, it consumes the next argument, foo.txt, and stores it in options.filename. So, after this call to parse_args(), options.filename is "foo.txt".

Some other option types supported by optparse are int and float. Here's an option that expects an integer argument:

```
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is store.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option: since -n42 (one argument) is equivalent to -n 42 (two arguments), the code

```
(options, args) = parser.parse_args(["-n42"])
print options.num
```

will print 42.

If you don't specify a type, optparse assumes string. Combined with the fact that the default action is store, that means our first example can be a lot shorter:

```
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, optparse figures out a sensible default from the option strings: if the first long option string is --foo-bar, then the default destination is foo_bar. If there are no long option strings, optparse looks at the first short option string: the default destination for -f is f.

optparse also includes built-in long and complex types. Adding types is covered in section Extending optparse.

Handling boolean (flag) options

Flag options—set a variable to true or false when a particular option is seen —are quite common. optparse supports them with two separate actions, store_true and store_false. For example, you might have a verbose flag that is turned on with -v and off with -q:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values— see below.)

When optparse encounters -v on the command line, it sets options.verbose to True; when it encounters -q, options.verbose is set to False.

Other actions

Some other actions supported by optparse are:

"store_const" store a constant value

"append" append this option's argument to a list

"count" increment a counter by one

"callback" call a specified function

These are covered in section Reference Guide, Reference Guide and section Option Callbacks.

Default values

All of the above examples involve setting some variable (the "destination") when certain command-line options are seen. What happens if those options are never seen? Since we didn't supply any defaults, they are all set to None. This is usually fine, but sometimes you want more control. optparse lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want optparse to set verbose to True unless -q is seen, then we can do this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the destination rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Consider this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for verbose will be True: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the set_defaults() method of OptionParser, which you can call at any time before calling parse_args():

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

Generating help

optparse's ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a help value for each option, and optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

If optparse encounters either -h or --help on the command-line, or if you just call parser.print_help(), it prints the following to standard output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(If the help output is triggered by a help option, optparse exits after printing the help text.)

There's a lot going on here to help optparse generate the best possible help message:

- the script defines its own usage message:

  ```
  usage = "usage: %prog [options] arg1 arg2"
  ```

  optparse expands %prog in the usage string to the name of the current program, i.e. os.path.basename(sys.argv[0]). The expanded string is then printed before the detailed option help.

  If you don't supply a usage string, optparse uses a bland but sensible default: "Usage: %prog [options]", which is fine if your script doesn't take any positional arguments.

- every option defines a help string, and doesn't worry about line-wrapping— optparse takes care of wrapping lines and making the help output look good.

- options that take a value indicate this fact in their automatically-generated help message, e.g. for the "mode" option:

  ```
  -m MODE, --mode=MODE
  ```

  Here, "MODE" is called the meta-variable: it stands for the argument that the user is expected to supply to -m/--mode. By default, optparse converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that's not what you want—for example, the --filename option explicitly sets metavar="FILE", resulting in this automatically-generated option description:

  ```
  -f FILE, --filename=FILE
  ```

  This is important for more than just saving space, though: the manually written help text uses the meta-variable FILE to clue the user in that there's a connection between the semi-formal syntax -f FILE and the informal semantic description "write output to FILE". This is a simple but effective way to make your help text a lot clearer and more useful for end users.

New in version 2.4: Options that have a default value can include %default in the help string—optparse will replace it with str() of the option's default value. If an option has no default value (or the default value is None), %default expands to none.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An OptionParser can contain several option groups, each of which can contain several options.

An option group is obtained using the class OptionGroup:

class optparse.OptionGroup(parser, title, description=None)
where

- parser is the OptionParser instance the group will be insterted in to
- title is the group title
- description, optional, is a long description of the group

OptionGroup inherits from OptionContainer (like OptionParser) and so the add_option() method can be used to add an option to the group.

Once all the options are declared, using the OptionParser method add_option_group() the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an OptionGroup to a parser is easy:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

This would result in the following help output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

A bit more complete example might involve using more than one group: still extending the previous example:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

that results in the following output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.

  Debug Options:
    -d, --debug         Print debug information
    -s, --sql           Print all SQL statements executed
    -e                  Print every action done
```

Another interesting method, in particular when working programmatically with option groups is:

OptionParser.get_option_group(opt_str)
: Return the OptionGroup to which the short or long option string opt_str (e.g. `'-o'` or `'--option'`) belongs. If there's no such OptionGroup, return None.

Printing a version string

Similar to the brief usage string, optparse can also print a version string for your program. You have to supply the string as the version argument to OptionParser:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

%prog is expanded just like it is in usage. Apart from that, version can contain anything you like. When you supply it, optparse automatically adds a --version option to your parser. If it encounters this option on the command line, it expands your version string (by replacing %prog), prints it to stdout, and exits.

For example, if your script is called /usr/bin/foo:

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the version string:

OptionParser.print_version(file=None)
: Print the version message for the current program (self.version) to file (default stdout). As with print_usage(), any occurrence of %prog in self.version is replaced with the name of the current program. Does nothing if self.version is empty or undefined.

OptionParser.get_version()
: Same as print_version() but returns the version string instead of printing it.

How optparse handles errors

There are two broad classes of errors that optparse has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to OptionParser.add_option(), e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either optparse.OptionError or TypeError) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. optparse can automatically detect some user errors, such as bad option arguments (passing -n 4x where -n takes an integer argument), missing arguments (-n at the end of the command line, where -n takes an argument of any type). Also, you can call OptionParser.error() to signal an application-defined error condition:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, optparse handles the error the same way: it prints the program's usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes 4x to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling OptionParser.error() from your application code.

If optparse's default error-handling behaviour does not suit your needs, you'll need to subclass OptionParser and override its exit() and/or error() methods.

Putting it all together

Here's what optparse-based scripts usually look like:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
```

(continues on next page)

(continued from previous page)

```
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print "reading %s..." % options.filename
    ...

if __name__ == "__main__":
    main()
```

15.5.3 Reference Guide

Creating the parser

The first step in using optparse is to create an OptionParser instance.

class optparse.OptionParser(...)
: The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

  usage (default: "%prog [options]") The usage summary to print when your program is run incorrectly or with a help option. When optparse prints the usage string, it expands %prog to os.path.basename(sys.argv[0]) (or to prog if you passed that keyword argument). To suppress a usage message, pass the special value optparse.SUPPRESS_USAGE.

  option_list (default: []) A list of Option objects to populate the parser with. The options in option_list are added after any options in standard_option_list (a class attribute that may be set by OptionParser subclasses), but before any version or help options. Deprecated; use add_option() after creating the parser instead.

  option_class (default: optparse.Option) Class to use when adding options to the parser in add_option().

  version (default: None) A version string to print when the user supplies a version option. If you supply a true value for version, optparse automatically adds a version option with the single option string --version. The substring %prog is expanded the same as for usage.

  conflict_handler (default: "error") Specifies what to do when options with conflicting option strings are added to the parser; see section Conflicts between options.

  description (default: None) A paragraph of text giving a brief overview of your program. optparse reformats this paragraph to fit the current terminal width and prints it when the user requests help (after usage, but before the list of options).

  formatter (default: a new IndentedHelpFormatter) An instance of optparse.HelpFormatter that will be used for printing help text. optparse provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

  add_help_option (default: True) If true, optparse will add a help option (with option strings -h and --help) to the parser.

  prog The string to use when expanding %prog in usage and version instead of os.path.basename(sys.argv[0]).

  epilog (default: None) A paragraph of help text to print after the option help.

Populating the parser

There are several ways to populate the parser with options. The preferred way is by using OptionParser.add_option(), as shown in section Tutorial. add_option() can be called in one of two ways:

- pass it an Option instance (as returned by make_option())
- pass it any combination of positional and keyword arguments that are acceptable to make_option() (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```
option_list = [
   make_option("-f", "--filename",
               action="store", type="string", dest="filename"),
   make_option("-q", "--quiet",
               action="store_false", dest="verbose"),
   ]
parser = OptionParser(option_list=option_list)
```

(make_option() is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of optparse may split Option into several classes, and make_option() will pick the right class to instantiate. Do not instantiate Option directly.)

Defining options

Each Option instance represents a set of synonymous command-line option strings, e.g. -f and --file. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an Option instance is with the add_option() method of OptionParser.

OptionParser.add_option(option)
OptionParser.add_option(*opt_str, attr=value, ...)

To define an option with only a short option string:

```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is action, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, optparse raises an OptionError exception explaining your mistake.

An option's action determines what optparse does when it encounters this option on the command-line. The standard option actions hard-coded into optparse are:

"store" store this option's argument (default)

"store_const" store a constant value

"store_true" store a true value

"store_false" store a false value

"append" append this option's argument to a list

"append_const" append a constant value to a list

"count" increment a counter by one

"callback" call a specified function

"help" print a usage message including all options and the documentation for them

(If you don't supply an action, the default is "store". For this action, you may also supply type and dest option attributes; see Standard option actions.)

As you can see, most actions involve storing or updating a value somewhere. optparse always creates a special object for this, conventionally called options (it happens to be an instance of optparse.Values). Option arguments (and various other values) are stored as attributes of this object, according to the dest (destination) option attribute.

For example, when you call

```
parser.parse_args()
```

one of the first things optparse does is create the options object:

```
options = Values()
```

If one of the options in this parser is defined with

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

and the command-line being parsed includes any of the following:

```
-ffoo
-f foo
--file=foo
--file foo
```

then optparse, on seeing this option, will do the equivalent of

```
options.filename = "foo"
```

The type and dest option attributes are almost as important as action, but action is the only one that makes sense for all options.

Option attributes

The following option attributes may be passed as keyword arguments to OptionParser.add_option(). If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, optparse raises OptionError.

Option.action
(default: "store")

Determines optparse's behaviour when this option is seen on the command line; the available options are documented here.

Option.type
(default: "string")

The argument type expected by this option (e.g., "string" or "int"); the available option types are documented here.

Option.dest
(default: derived from option strings)

If the option's action implies writing or modifying a value somewhere, this tells optparse where to write it: dest names an attribute of the options object that optparse builds as it parses the command line.

Option.default
The value to use for this option's destination if the option is not seen on the command line. See also OptionParser.set_defaults().

Option.nargs
(default: 1)

How many arguments of type type should be consumed when this option is seen. If > 1, optparse will store a tuple of values to dest.

Option.const
For actions that store a constant value, the constant value to store.

Option.choices
For options of type "choice", the list of strings the user may choose from.

Option.callback
For options with action "callback", the callable to call when this option is seen. See section Option Callbacks for detail on the arguments passed to the callable.

Option.callback_args
Option.callback_kwargs
Additional positional and keyword arguments to pass to callback after the four standard callback arguments.

Option.help
Help text to print for this option when listing all available options after the user supplies a help option (such as --help). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value optparse.SUPPRESS_HELP.

Option.metavar
(default: derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section Tutorial for an example.

Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide optparse's behaviour; a few have required attributes, which you must specify for any option using that action.

- "store" [relevant: type, dest, nargs, choices]

  The option must be followed by an argument, which is converted to a value according to type and stored in dest. If nargs > 1, multiple arguments will be consumed from the command line; all will be converted according to type and stored to dest as a tuple. See the Standard option types section.

  If choices is supplied (a list or tuple of strings), the type defaults to "choice".

  If type is not supplied, it defaults to "string".

  If dest is not supplied, optparse derives a destination from the first long option string (e.g., --foo-bar implies foo_bar). If there are no long option strings, optparse derives a destination from the first short option string (e.g., -f implies f).

  Example:

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

As it parses the command line

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

optparse will set

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- "store_const" [required: const; relevant: dest]

  The value const is stored in dest.

  Example:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

  If --noisy is seen, optparse will set

```
options.verbose = 2
```

- "store_true" [relevant: dest]

  A special case of "store_const" that stores a true value to dest.

- "store_false" [relevant: dest]

  Like "store_true", but stores a false value.

  Example:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [relevant: type, dest, nargs, choices]

  The option must be followed by an argument, which is appended to the list in dest. If no default value for dest is supplied, an empty list is automatically created when optparse first encounters this option on the command-line. If nargs > 1, multiple arguments are consumed, and a tuple of length nargs is appended to dest.

  The defaults for type and dest are the same as for the "store" action.

  Example:

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

  If -t3 is seen on the command-line, optparse does the equivalent of:

```
options.tracks = []
options.tracks.append(int("3"))
```

If, a little later on, --tracks=4 is seen, it does:

```
options.tracks.append(int("4"))
```

The append action calls the append method on the current value of the option. This means that any default value specified must have an append method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:

```
>>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/.mypkg/defaults', 'overrides.mypkg']
```

- "append_const" [required: const; relevant: dest]

  Like "store_const", but the value const is appended to dest; as with "append", dest defaults to None, and an empty list is automatically created the first time the option is encountered.

- "count" [relevant: dest]

  Increment the integer stored at dest. If no default value is supplied, dest is set to zero before being incremented the first time.

  Example:

  ```
  parser.add_option("-v", action="count", dest="verbosity")
  ```

  The first time -v is seen on the command line, optparse does the equivalent of:

  ```
  options.verbosity = 0
  options.verbosity += 1
  ```

  Every subsequent occurrence of -v results in

  ```
  options.verbosity += 1
  ```

- "callback" [required: callback; relevant: type, nargs, callback_args, callback_kwargs]

  Call the function specified by callback, which is called as

  ```
  func(option, opt_str, value, parser, *args, **kwargs)
  ```

  See section Option Callbacks for more detail.

- "help"

  Prints a complete help message for all the options in the current option parser. The help message is constructed from the usage string passed to OptionParser's constructor and the help string passed to every option.

  If no help string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value optparse.SUPPRESS_HELP.

  optparse automatically adds a help option to all OptionParsers, so you do not normally need to create one.

  Example:

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can
# be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

If optparse sees either -h or --help on the command line, it will print something like the following help message to stdout (assuming sys.argv[0] is "foo.py"):

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

After printing the help message, optparse terminates your process with sys.exit(0).

- "version"

  Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the print_version() method of OptionParser. Generally only relevant if the version argument is supplied to the OptionParser constructor. As with help options, you will rarely create version options, since optparse automatically adds them when needed.

Standard option types

optparse has six built-in option types: "string", "int", "long", "choice", "float" and "complex". If you need to add new option types, see section Extending optparse.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type "int" or "long") are parsed as follows:

- if the number starts with 0x, it is parsed as a hexadecimal number
- if the number starts with 0, it is parsed as an octal number
- if the number starts with 0b, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling either int() or long() with the appropriate base (2, 8, 10, or 16). If this fails, so will optparse, although with a more useful error message.

"float" and "complex" option arguments are converted directly with float() and complex(), with similar error-handling.

"choice" options are a subtype of "string" options. The choices option attribute (a sequence of strings) defines the set of allowed option arguments. optparse.check_choice() compares user-supplied option arguments against this master list and raises OptionValueError if an invalid string is given.

Parsing arguments

The whole point of creating and populating an OptionParser is to call its parse_args() method:

```
(options, args) = parser.parse_args(args=None, values=None)
```

where the input parameters are

args the list of arguments to process (default: sys.argv[1:])

values an optparse.Values object to store option arguments in (default: a new instance of Values) – if you give an existing object, the option defaults will not be initialized on it

and the return values are

options the same object that was passed in as values, or the optparse.Values instance created by optparse

args the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply values, it will be modified with repeated setattr() calls (roughly one for every option argument stored to an option destination) and returned by parse_args().

If parse_args() encounters any errors in the argument list, it calls the OptionParser's error() method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what's there. OptionParser provides several methods to help you out:

OptionParser.disable_interspersed_args()
Set parsing to stop on the first non-option. For example, if -a and -b are both simple options that take no arguments, optparse normally accepts this syntax:

```
prog -a arg1 -b arg2
```

and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call disable_interspersed_args(). This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don't get confused. For example, each command might have a different set of options.

OptionParser.enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

OptionParser.get_option(opt_str)
Returns the Option instance with the option string opt_str, or None if no options have that option string.

OptionParser.has_option(opt_str)
Return true if the OptionParser has an option with option string opt_str (e.g., -q or --verbose).

OptionParser.remove_option(opt_str)
: If the OptionParser has an option corresponding to opt_str, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If opt_str does not occur in any option belonging to this OptionParser, raises ValueError.

Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you've defined your own OptionParser subclass with some standard options.)

Every time you add an option, optparse checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```
parser.set_conflict_handler(handler)
```

The available conflict handlers are:

"error" (default) assume option conflicts are a programming error and raise OptionConflictError

"resolve" resolve option conflicts intelligently (see below)

As an example, let's define an OptionParser that resolves conflicts intelligently and add conflicting options to it:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, optparse detects that a previously-added option is already using the -n option string. Since conflict_handler is "resolve", it resolves the situation by removing -n from the earlier option's list of option strings. Now --dry-run is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

It's possible to whittle away the option strings for a previously-added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, optparse removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing OptionParser:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original -n/--dry-run option is no longer accessible, so optparse removes it, leaving this help text:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

Cleanup

OptionParser instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling destroy() on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

Other methods

OptionParser supports several other public methods:

OptionParser.set_usage(usage)
: Set the usage string according to the rules described above for the usage constructor keyword argument. Passing None sets the default usage string; use optparse.SUPPRESS_USAGE to suppress a usage message.

OptionParser.print_usage(file=None)
: Print the usage message for the current program (self.usage) to file (default stdout). Any occurrence of the string %prog in self.usage is replaced with the name of the current program. Does nothing if self.usage is empty or not defined.

OptionParser.get_usage()
: Same as print_usage() but returns the usage string instead of printing it.

OptionParser.set_defaults(dest=value, ...)
: Set default values for several option destinations at once. Using set_defaults() is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several "mode" options all set the same destination, any one of them can set the default, and the last one wins:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # overrides above setting
```

To avoid this confusion, use set_defaults():

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

15.5.4 Option Callbacks

When optparse's built-in actions and types aren't quite enough for your needs, you have two choices: extend optparse or define a callback option. Extending optparse is more general, but overkill for a lot of simple

cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the "callback" action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the OptionParser.add_option() method. Apart from action, the only option attribute you must specify is callback, the function to call:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

callback is a function (or other callable object), so you must have already defined my_callback() when you create this callback option. In this simple case, optparse doesn't even know if -c takes any arguments, which usually means that the option takes no arguments—the mere presence of -c on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this section.

optparse always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via callback_args and callback_kwargs. Thus, the minimal callback function signature is:

```
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

type has its usual meaning: as with the "store" or "append" actions, it instructs optparse to consume one argument and convert it to type. Rather than storing the converted value(s) anywhere, though, optparse passes it to your callback function.

nargs also has its usual meaning: if it is supplied and > 1, optparse will consume nargs arguments, each of which must be convertible to type. It then passes a tuple of converted values to your callback.

callback_args a tuple of extra positional arguments to pass to the callback

callback_kwargs a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

option is the Option instance that's calling the callback

opt_str is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, opt_str will be the full, canonical option string—e.g. if the user puts --foo on the command-line as an abbreviation for --foobar, then opt_str will be "--foobar".)

value is the argument to this option seen on the command-line. optparse will only expect an argument if type is set; the type of value will be the type implied by the option's type. If type for this option is None (no argument expected), then value will be None. If nargs > 1, value will be a tuple of values of the appropriate type.

parser is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

parser.largs the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor option arguments. Feel free to modify parser.largs, e.g. by adding more arguments to it. (This list will become args, the second return value of parse_args().)

parser.rargs the current list of remaining arguments, ie. with opt_str and value (if applicable) removed, and only the arguments following them still there. Feel free to modify parser.rargs, e.g. by consuming more arguments.

parser.values the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use the same mechanism as the rest of optparse for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

args is a tuple of arbitrary positional arguments supplied via the callback_args option attribute.

kwargs is a dictionary of arbitrary keyword arguments supplied via callback_kwargs.

Raising errors in a callback

The callback function should raise OptionValueError if there are any problems with the option or its argument(s). optparse catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

Callback example 1: trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the "store_true" action.

Callback example 2: check option order

Here's a slightly more interesting example: record the fact that -a is seen, but blow up if it comes after -b in the command-line.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

Callback example 3: check option order (generalized)

If you want to re-use this callback for several similar options (set a flag, but blow up if -b has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

Callback example 4: check arbitrary condition

Of course, you could put any condition in there—you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(The definition of is_moon_full() is left as an exercise for the reader.)

Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a "store" or "append" option: if you define type, then the option takes one argument that must be convertible to that type; if you further define nargs, then the option takes nargs arguments.

Here's an example that just emulates the standard "store" action:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

Note that optparse takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don't need a callback for this example.)

Callback example 6: variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as optparse doesn't provide any built-in capabilities for it. And you have to deal with

certain intricacies of conventional Unix command-line parsing that optparse normally handles for you. In particular, callbacks should implement the conventional rules for bare -- and - arguments:

- either -- or - can be option arguments
- bare -- (if not the argument to some option): halt command-line processing and discard the --
- bare - (if not the argument to some option): halt command-line processing but keep the - (append it to parser.largs)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why optparse doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments:

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)

    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

15.5.5 Extending optparse

Since the two major controlling factors in how optparse interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of optparse's Option class. This class has a couple of attributes that define optparse's types: TYPES and TYPE_CHECKER.

Option.TYPES
: A tuple of type names; in your subclass, simply define a new tuple TYPES that builds on the standard one.

Option.TYPE_CHECKER

A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```
def check_mytype(option, opt, value)
```

where option is an Option instance, opt is an option string (e.g., -f), and value is the string from the command line that must be checked and converted to your desired type. check_mytype() should return an object of the hypothetical type mytype. The value returned by a type-checking function will wind up in the OptionValues instance returned by OptionParser.parse_args(), or be passed to a callback as the value parameter.

Your type-checking function should raise OptionValueError if it encounters any problems. OptionValueError takes a single string argument, which is passed as-is to OptionParser's error() method, which in turn prepends the program name and the string "error:" and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a "complex" option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because optparse 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the TYPE_CHECKER class attribute of your Option subclass):

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finally, the Option subclass:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a copy() of Option.TYPE_CHECKER, we would end up modifying the TYPE_CHECKER attribute of optparse's Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That's it! Now you can write a script that uses the new option type just like any other optparse-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser; if you don't use add_option() in the above way, you don't need to tell OptionParser which option class to use:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

Adding new actions

Adding new actions is a bit trickier, because you have to understand that optparse has a couple of classifications for actions:

"store" actions actions that result in optparse storing a value to an attribute of the current OptionValues instance; these options require a dest attribute to be supplied to the Option constructor.

"typed" actions actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a type attribute to the Option constructor.

These are overlapping sets: some default "store" actions are "store", "store_const", "append", and "count", while the default "typed" actions are "store", "append", and "callback".

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings):

Option.ACTIONS
All actions must be listed in ACTIONS.

Option.STORE_ACTIONS
"store" actions are additionally listed here.

Option.TYPED_ACTIONS
"typed" actions are additionally listed here.

Option.ALWAYS_TYPED_ACTIONS
Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that optparse assigns the default type, "string", to options with no explicit type whose action is listed in ALWAYS_TYPED_ACTIONS.

In order to actually implement your new action, you must override Option's take_action() method and add a case that recognizes your action.

For example, let's add an "extend" action. This is similar to the standard "append" action, but instead of taking a single value from the command-line and appending it to an existing list, "extend" will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if --names is an "extend" option of type "string", the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
```

(continues on next page)

(continued from previous page)

```
Option.take_action(
    self, action, dest, opt, value, values, parser)
```

Features of note:

- "extend" both expects a value on the command-line and stores that value somewhere, so it goes in both STORE_ACTIONS and TYPED_ACTIONS.
- to ensure that optparse assigns the default type of "string" to "extend" actions, we put the "extend" action in ALWAYS_TYPED_ACTIONS as well.
- MyOption.take_action() implements just this one new action, and passes control back to Option.take_action() for the standard optparse actions.
- values is an instance of the optparse_parser.Values class, which provides the very useful ensure_value() method. ensure_value() is essentially getattr() with a safety valve; it is called as

  ```
  values.ensure_value(attr, value)
  ```

  If the attr attribute of values doesn't exist or is None, then ensure_value() first sets it to value, and then returns 'value. This is very handy for actions like "extend", "append", and "count", all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using ensure_value() means that scripts using your action don't have to worry about setting a default value for the option destinations in question; they can just leave the default as None and ensure_value() will take care of getting it right when it's needed.

15.6 getopt — C-style parser for command line options

Source code: Lib/getopt.py

Note: The getopt module is a parser for command line options whose API is designed to be familiar to users of the C getopt() function. Users who are unfamiliar with the C getopt() function or who would like to write less code and get better help and error messages should consider using the argparse module instead.

This module helps scripts to parse the command line arguments in sys.argv. It supports the same conventions as the Unix getopt() function (including the special meanings of arguments of the form '-' and '--'). Long options similar to those supported by GNU software may be used as well via an optional third argument.

This module provides two functions and an exception:

getopt.getopt(args, options[, long_options])
: Parses command line options and parameter list. args is the argument list to be parsed, without the leading reference to the running program. Typically, this means sys.argv[1:]. options is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (':'; i.e., the same format that Unix getopt() uses).

  Note: Unlike GNU getopt(), after a non-option argument, all further arguments are considered also non-options. This is similar to the way non-GNU Unix systems work.

  long_options, if specified, must be a list of strings with the names of the long options which should be supported. The leading '--' characters should not be included in the option name. Long options

which require an argument should be followed by an equal sign ('='). Optional arguments are not supported. To accept only long options, options should be an empty string. Long options on the command line can be recognized so long as they provide a prefix of the option name that matches exactly one of the accepted options. For example, if long_options is ['foo', 'frob'], the option --fo will match as --foo, but --f will not match uniquely, so GetoptError will be raised.

The return value consists of two elements: the first is a list of (option, value) pairs; the second is the list of program arguments left after the option list was stripped (this is a trailing slice of args). Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option'), and the option argument as its second element, or an empty string if the option has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Long and short options may be mixed.

getopt.gnu_getopt(args, options[, long_options])
This function works like getopt(), except that GNU style scanning mode is used by default. This means that option and non-option arguments may be intermixed. The getopt() function stops processing options as soon as a non-option argument is encountered.

If the first character of the option string is '+', or if the environment variable POSIXLY_CORRECT is set, then option processing stops as soon as a non-option argument is encountered.

New in version 2.3.

exception getopt.GetoptError
This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error. For long options, an argument given to an option which does not require one will also cause this exception to be raised. The attributes msg and opt give the error message and related option; if there is no specific option to which the exception relates, opt is an empty string.

Changed in version 1.6: Introduced GetoptError as a synonym for error.

exception getopt.error
Alias for GetoptError; for backward compatibility.

An example using only Unix style options:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

Using long option names is equally easy:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '')]
```

(continues on next page)

(continued from previous page)

```
>>> args
['a1', 'a2']
```

In a script, typical usage is something like this:

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print str(err)  # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    # ...

if __name__ == "__main__":
    main()
```

Note that an equivalent command line interface could be produced with less code and more informative help and error messages by using the argparse module:

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ..
```

See also:

Module argparse Alternative command line option and argument parsing library.

15.7 logging — Logging facility for Python

Important

This page contains the API reference information. For tutorial information and discussion of more advanced topics, see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

Source code: Lib/logging/__init__.py

New in version 2.3.

This module defines functions and classes which implement a flexible event logging system for applications and libraries.

The key benefit of having the logging API provided by a standard library module is that all Python modules can participate in logging, so your application log can include your own messages integrated with messages from third-party modules.

The module provides a lot of functionality and flexibility. If you are unfamiliar with logging, the best way to get to grips with it is to see the tutorials (see the links on the right).

The basic classes defined by the module, together with their functions, are listed below.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.

15.7.1 Logger Objects

Loggers have the following attributes and methods. Note that Loggers are never instantiated directly, but always through the module-level function logging.getLogger(name). Multiple calls to getLogger() with the same name will always return a reference to the same Logger object.

The name is potentially a period-separated hierarchical value, like foo.bar.baz (though it could also be just plain foo, for example). Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of foo, loggers with names of foo.bar, foo.bar.baz, and foo.bam are all descendants of foo. The logger name hierarchy is analogous to the Python package hierarchy, and identical to it if you organise your loggers on a per-module basis using the recommended construction logging.getLogger(__name__). That's because in a module, __name__ is the module's name in the Python package namespace.

class logging.Logger

Logger.propagate
: If this evaluates to true, events logged to this logger will be passed to the handlers of higher level (ancestor) loggers, in addition to any handlers attached to this logger. Messages are passed directly to the ancestor loggers' handlers - neither the level nor filters of the ancestor loggers in question are considered.

  If this evaluates to false, logging messages are not passed to the handlers of ancestor loggers.

  The constructor sets this attribute to True.

Note: If you attach a handler to a logger and one or more of its ancestors, it may emit the same record multiple times. In general, you should not need to attach a handler to more than one logger - if you just attach it to the appropriate logger which is highest in the logger hierarchy, then it will see all events logged by all descendant loggers, provided that their propagate setting is left set to True. A common scenario is to attach handlers only to the root logger, and to let propagation take care of the rest.

Logger.setLevel(level)
Sets the threshold for this logger to level. Logging messages which are less severe than level will be ignored. When a logger is created, the level is set to NOTSET (which causes all messages to be processed when the logger is the root logger, or delegation to the parent when the logger is a non-root logger). Note that the root logger is created with level WARNING.

The term 'delegation to the parent' means that if a logger has a level of NOTSET, its chain of ancestor loggers is traversed until either an ancestor with a level other than NOTSET is found, or the root is reached.

If an ancestor is found with a level other than NOTSET, then that ancestor's level is treated as the effective level of the logger where the ancestor search began, and is used to determine how a logging event is handled.

If the root is reached, and it has a level of NOTSET, then all messages will be processed. Otherwise, the root's level will be used as the effective level.

See Logging Levels for a list of levels.

Logger.isEnabledFor(lvl)
Indicates if a message of severity lvl would be processed by this logger. This method checks first the module-level level set by logging.disable(lvl) and then the logger's effective level as determined by getEffectiveLevel().

Logger.getEffectiveLevel()
Indicates the effective level for this logger. If a value other than NOTSET has been set using setLevel(), it is returned. Otherwise, the hierarchy is traversed towards the root until a value other than NOTSET is found, and that value is returned. The value returned is an integer, typically one of logging.DEBUG, logging.INFO etc.

Logger.getChild(suffix)
Returns a logger which is a descendant to this logger, as determined by the suffix. Thus, logging.getLogger('abc').getChild('def.ghi') would return the same logger as would be returned by logging.getLogger('abc.def.ghi'). This is a convenience method, useful when the parent logger is named using e.g. __name__ rather than a literal string.

New in version 2.7.

Logger.debug(msg, *args, **kwargs)
Logs a message with level DEBUG on this logger. The msg is the message format string, and the args are the arguments which are merged into msg using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are two keyword arguments in kwargs which are inspected: exc_info which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by sys.exc_info()) is provided, it is used; otherwise, sys.exc_info() is called to get the exception information.

The second keyword argument is extra which can be used to pass a dictionary which is used to populate the __dict__ of the LogRecord created for the logging event with user-defined attributes.

These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

would print something like

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

The keys in the dictionary passed in extra should not clash with the keys used by the logging system. (See the Formatter documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the Formatter has been set up with a format string which expects 'clientip' and 'user' in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the extra dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized Formatters would be used with particular Handlers.

Logger.info(msg, *args, **kwargs)
: Logs a message with level INFO on this logger. The arguments are interpreted as for debug().

Logger.warning(msg, *args, **kwargs)
: Logs a message with level WARNING on this logger. The arguments are interpreted as for debug().

Logger.error(msg, *args, **kwargs)
: Logs a message with level ERROR on this logger. The arguments are interpreted as for debug().

Logger.critical(msg, *args, **kwargs)
: Logs a message with level CRITICAL on this logger. The arguments are interpreted as for debug().

Logger.log(lvl, msg, *args, **kwargs)
: Logs a message with integer level lvl on this logger. The other arguments are interpreted as for debug().

Logger.exception(msg, *args, **kwargs)
: Logs a message with level ERROR on this logger. The arguments are interpreted as for debug(), except that any passed exc_info is not inspected. Exception info is always added to the logging message. This method should only be called from an exception handler.

Logger.addFilter(filter)
: Adds the specified filter filter to this logger.

Logger.removeFilter(filter)
: Removes the specified filter filter from this logger.

Logger.filter(record)
: Applies this logger's filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be processed (passed to handlers). If one returns a false value, no further processing of the record occurs.

Logger.addHandler(hdlr)
Adds the specified handler hdlr to this logger.

Logger.removeHandler(hdlr)
Removes the specified handler hdlr from this logger.

Logger.findCaller()
Finds the caller's source filename and line number. Returns the filename, line number and function name as a 3-element tuple.

Changed in version 2.4: The function name was added. In earlier versions, the filename and line number were returned as a 2-element tuple.

Logger.handle(record)
Handles a record by passing it to all handlers associated with this logger and its ancestors (until a false value of propagate is found). This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is applied using filter().

Logger.makeRecord(name, lvl, fn, lno, msg, args, exc_info, func=None, extra=None)
This is a factory method which can be overridden in subclasses to create specialized LogRecord instances.

Changed in version 2.5: func and extra were added.

15.7.2 Logging Levels

The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

Level	Numeric value
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

15.7.3 Handler Objects

Handlers have the following attributes and methods. Note that Handler is never instantiated directly; this class acts as a base for more useful subclasses. However, the __init__() method in subclasses needs to call Handler.__init__().

Handler.__init__(level=NOTSET)
Initializes the Handler instance by setting its level, setting the list of filters to the empty list and creating a lock (using createLock()) for serializing access to an I/O mechanism.

Handler.createLock()
Initializes a thread lock which can be used to serialize access to underlying I/O functionality which may not be threadsafe.

Handler.acquire()
Acquires the thread lock created with createLock().

Handler.release()
: Releases the thread lock acquired with acquire().

Handler.setLevel(level)
: Sets the threshold for this handler to level. Logging messages which are less severe than level will be ignored. When a handler is created, the level is set to NOTSET (which causes all messages to be processed).

  See Logging Levels for a list of levels.

Handler.setFormatter(fmt)
: Sets the Formatter for this handler to fmt.

Handler.addFilter(filter)
: Adds the specified filter filter to this handler.

Handler.removeFilter(filter)
: Removes the specified filter filter from this handler.

Handler.filter(record)
: Applies this handler's filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be emitted. If one returns a false value, the handler will not emit the record.

Handler.flush()
: Ensure all logging output has been flushed. This version does nothing and is intended to be implemented by subclasses.

Handler.close()
: Tidy up any resources used by the handler. This version does no output but removes the handler from an internal list of handlers which is closed when shutdown() is called. Subclasses should ensure that this gets called from overridden close() methods.

Handler.handle(record)
: Conditionally emits the specified logging record, depending on filters which may have been added to the handler. Wraps the actual emission of the record with acquisition/release of the I/O thread lock.

Handler.handleError(record)
: This method should be called from handlers when an exception is encountered during an emit() call. If the module-level attribute raiseExceptions is False, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The specified record is the one which was being processed when the exception occurred. (The default value of raiseExceptions is True, as that is more useful during development).

Handler.format(record)
: Do formatting for a record - if a formatter is set, use it. Otherwise, use the default formatter for the module.

Handler.emit(record)
: Do whatever it takes to actually log the specified logging record. This version is intended to be implemented by subclasses and so raises a NotImplementedError.

For a list of handlers included as standard, see logging.handlers.

15.7.4 Formatter Objects

Formatter objects have the following attributes and methods. They are responsible for converting a LogRecord to (usually) a string which can be interpreted by either a human or an external system. The base Formatter allows a formatting string to be specified. If none is supplied, the default value of `'%(message)s'`

is used, which just includes the message in the logging call. To have additional items of information in the formatted output (such as a timestamp), keep reading.

A Formatter can be initialized with a format string which makes use of knowledge of the LogRecord attributes - such as the default value mentioned above making use of the fact that the user's message and arguments are pre-formatted into a LogRecord's message attribute. This format string contains standard Python %-style mapping keys. See section String Formatting Operations for more information on string formatting.

The useful mapping keys in a LogRecord are given in the section on LogRecord attributes.

class logging.Formatter(fmt=None, datefmt=None)
Returns a new instance of the Formatter class. The instance is initialized with a format string for the message as a whole, as well as a format string for the date/time portion of a message. If no fmt is specified, `'%(message)s'` is used. If no datefmt is specified, the ISO8601 date format is used.

format(record)
The record's attribute dictionary is used as the operand to a string formatting operation. Returns the resulting string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using msg % args. If the formatting string contains `'(asctime)'`, formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message. Note that the formatted exception information is cached in attribute exc_text. This is useful because the exception information can be pickled and sent across the wire, but you should be careful if you have more than one Formatter subclass which customizes the formatting of exception information. In this case, you will have to clear the cached value after a formatter has done its formatting, so that the next formatter to handle the event doesn't use the cached value but recalculates it afresh.

formatTime(record, datefmt=None)
This method should be called from format() by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behavior is as follows: if datefmt (a string) is specified, it is used with time.strftime() to format the creation time of the record. Otherwise, the ISO8601 format is used. The resulting string is returned.

This function uses a user-configurable function to convert the creation time to a tuple. By default, time.localtime() is used; to change this for a particular formatter instance, set the converter attribute to a function with the same signature as time.localtime() or time.gmtime(). To change it for all formatters, for example if you want all logging times to be shown in GMT, set the converter attribute in the Formatter class.

formatException(exc_info)
Formats the specified exception information (a standard exception tuple as returned by sys.exc_info()) as a string. This default implementation just uses traceback.print_exception(). The resulting string is returned.

15.7.5 Filter Objects

Filters can be used by Handlers and Loggers for more sophisticated filtering than is provided by levels. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with 'A.B' will allow events logged by loggers 'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D' etc. but not 'A.BB', 'B.A.B' etc. If initialized with the empty string, all events are passed.

class logging.Filter(name='')
Returns an instance of the Filter class. If name is specified, it names a logger which, together with its children, will have its events allowed through the filter. If name is the empty string, allows every event.

filter(record)
Is the specified record to be logged? Returns zero for no, nonzero for yes. If deemed appropriate, the record may be modified in-place by this method.

Note that filters attached to handlers are consulted before an event is emitted by the handler, whereas filters attached to loggers are consulted whenever an event is logged (using debug(), info(), etc.), before sending an event to handlers. This means that events which have been generated by descendant loggers will not be filtered by a logger's filter setting, unless the filter has also been applied to those descendant loggers.

You don't actually need to subclass Filter: you can pass any instance which has a filter method with the same semantics.

Although filters are used primarily to filter records based on more sophisticated criteria than levels, they get to see every record which is processed by the handler or logger they're attached to: this can be useful if you want to do things like counting how many records were processed by a particular logger or handler, or adding, changing or removing attributes in the LogRecord being processed. Obviously changing the LogRecord needs to be done with some care, but it does allow the injection of contextual information into logs (see filters-contextual).

15.7.6 LogRecord Objects

LogRecord instances are created automatically by the Logger every time something is logged, and can be created manually via makeLogRecord() (for example, from a pickled event received over the wire).

class logging.LogRecord(name, level, pathname, lineno, msg, args, exc_info, func=None)
Contains all the information pertinent to the event being logged.

The primary information is passed in msg and args, which are combined using msg % args to create the message field of the record.

Parameters

- name – The name of the logger used to log the event represented by this LogRecord. Note that this name will always have this value, even though it may be emitted by a handler attached to a different (ancestor) logger.
- level – The numeric level of the logging event (one of DEBUG, INFO etc.) Note that this is converted to two attributes of the LogRecord: levelno for the numeric value and levelname for the corresponding level name.
- pathname – The full pathname of the source file where the logging call was made.
- lineno – The line number in the source file where the logging call was made.
- msg – The event description message, possibly a format string with placeholders for variable data.
- args – Variable data to merge into the msg argument to obtain the event description.
- exc_info – An exception tuple with the current exception information, or None if no exception information is available.
- func – The name of the function or method from which the logging call was invoked.

Changed in version 2.5: func was added.

getMessage()
Returns the message for this LogRecord instance after merging any user-supplied arguments with the message. If the user-supplied message argument to the logging call is not a string, str() is called on it to convert it to a string. This allows use of user-defined classes as messages, whose __str__ method can return the actual format string to be used.

15.7.7 LogRecord attributes

The LogRecord has a number of attributes, most of which are derived from the parameters to the constructor. (Note that the names do not always correspond exactly between the LogRecord constructor parameters and the LogRecord attributes.) These attributes can be used to merge data from the record into the format string. The following table lists (in alphabetical order) the attribute names, their meanings and the corresponding placeholder in a %-style format string.

Attribute name	Format	Description
args	You shouldn't need to format this yourself.	The tuple of arguments merged into msg to produce message, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	%(asctime)s	Human-readable time when the LogRecord was created. By default this is of the form '2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
created	%(created)f	Time when the LogRecord was created (as returned by time.time()).
exc_info	You shouldn't need to format this yourself.	Exception tuple (à la sys.exc_info) or, if no exception has occurred, None.
filename	%(filename)s	Filename portion of pathname.
funcName	%(funcName)s	Name of function containing the logging call.
levelname	%(levelname)s	Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL').
levelno	%(levelno)s	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL).
lineno	%(lineno)d	Source line number where the logging call was issued (if available).
module	%(module)s	Module (name portion of filename).
msecs	%(msecs)d	Millisecond portion of the time when the LogRecord was created.
message	%(message)s	The logged message, computed as msg % args. This is set when Formatter.format() is invoked.
msg	You shouldn't need to format this yourself.	The format string passed in the original logging call. Merged with args to produce message, or an arbitrary object (see arbitrary-object-messages).
name	%(name)s	Name of the logger used to log the call.
pathname	%(pathname)s	Full pathname of the source file where the logging call was issued (if available).
process	%(process)d	Process ID (if available).
processName	%(processName)s	Process name (if available).
relativeCreated	%(relativeCreated)d	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded.
thread	%(thread)d	Thread ID (if available).
threadName	%(threadName)s	Thread name (if available).

Changed in version 2.5: funcName was added.

Changed in version 2.6: processName was added.

15.7.8 LoggerAdapter Objects

LoggerAdapter instances are used to conveniently pass contextual information into logging calls. For a usage example, see the section on adding contextual information to your logging output.

New in version 2.6.

class logging.LoggerAdapter(logger, extra)
Returns an instance of LoggerAdapter initialized with an underlying Logger instance and a dict-like object.

process(msg, kwargs)
Modifies the message and/or keyword arguments passed to a logging call in order to insert contextual information. This implementation takes the object passed as extra to the constructor and adds it to kwargs using key 'extra'. The return value is a (msg, kwargs) tuple which has the (possibly modified) versions of the arguments passed in.

In addition to the above, LoggerAdapter supports the following methods of Logger: debug(), info(), warning(), error(), exception(), critical(), log() and isEnabledFor(). These methods have the same signatures as their counterparts in Logger, so you can use the two types of instances interchangeably for these calls.

Changed in version 2.7: The isEnabledFor() method was added to LoggerAdapter. This method delegates to the underlying logger.

15.7.9 Thread Safety

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this though using threading locks; there is one lock to serialize access to the module's shared data, and each handler also creates a lock to serialize access to its underlying I/O.

If you are implementing asynchronous signal handlers using the signal module, you may not be able to use logging from within such handlers. This is because lock implementations in the threading module are not always re-entrant, and so cannot be invoked from such signal handlers.

15.7.10 Module-Level Functions

In addition to the classes described above, there are a number of module- level functions.

logging.getLogger([name])
Return a logger with the specified name or, if no name is specified, return a logger which is the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like "a", "a.b" or "a.b.c.d". Choice of these names is entirely up to the developer who is using logging.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

logging.getLoggerClass()
Return either the standard Logger class, or the last class passed to setLoggerClass(). This function may be called from within a new class definition, to ensure that installing a customized Logger class will not undo customizations already applied by other code. For example:

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

logging.debug(msg[, *args[, **kwargs]])
Logs a message with level DEBUG on the root logger. The msg is the message format string, and the

args are the arguments which are merged into msg using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are two keyword arguments in kwargs which are inspected: exc_info which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by sys.exc_info()) is provided, it is used; otherwise, sys.exc_info() is called to get the exception information.

The other optional keyword argument is extra which can be used to pass a dictionary which is used to populate the __dict__ of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```
FORMAT = "%(asctime)-15s %(clientip)s %(user)-8s %(message)s"
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning("Protocol problem: %s", "connection reset", extra=d)
```

would print something like:

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

The keys in the dictionary passed in extra should not clash with the keys used by the logging system. (See the Formatter documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the Formatter has been set up with a format string which expects 'clientip' and 'user' in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the extra dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized Formatters would be used with particular Handlers.

Changed in version 2.5: extra was added.

logging.info(msg[, *args[, **kwargs]])
Logs a message with level INFO on the root logger. The arguments are interpreted as for debug().

logging.warning(msg[, *args[, **kwargs]])
Logs a message with level WARNING on the root logger. The arguments are interpreted as for debug().

logging.error(msg[, *args[, **kwargs]])
Logs a message with level ERROR on the root logger. The arguments are interpreted as for debug().

logging.critical(msg[, *args[, **kwargs]])
Logs a message with level CRITICAL on the root logger. The arguments are interpreted as for debug().

logging.exception(msg[, *args[, **kwargs]])
Logs a message with level ERROR on the root logger. The arguments are interpreted as for debug(), except that any passed exc_info is not inspected. Exception info is always added to the logging message. This function should only be called from an exception handler.

logging.log(level, msg[, *args[, **kwargs]])
Logs a message with level level on the root logger. The other arguments are interpreted as for debug().

Note: The above module-level convenience functions, which delegate to the root logger, call basicConfig() to ensure that at least one handler is available. Because of this, they should not be used in threads, in versions of Python earlier than 2.7.1 and 3.2, unless at least one handler has been added to the root logger before the threads are started. In earlier versions of Python, due to a thread safety shortcoming in basicConfig(), this can (under rare circumstances) lead to handlers being added multiple times to the root logger, which can in turn lead to multiple messages for the same event.

logging.disable(lvl)
Provides an overriding level lvl for all loggers which takes precedence over the logger's own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity lvl and below, so that if you call it with a value of INFO, then all INFO and DEBUG events would be discarded, whereas those of severity WARNING and above would be processed according to the logger's effective level. If logging.disable(logging.NOTSET) is called, it effectively removes this overriding level, so that logging output again depends on the effective levels of individual loggers.

logging.addLevelName(lvl, levelName)
Associates level lvl with text levelName in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a Formatter formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

Note: If you are thinking of defining your own levels, please see the section on custom-levels.

logging.getLevelName(lvl)
Returns the textual representation of logging level lvl. If the level is one of the predefined levels CRITICAL, ERROR, WARNING, INFO or DEBUG then you get the corresponding string. If you have associated levels with names using addLevelName() then the name you have associated with lvl is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned. Otherwise, the string "Level %s" % lvl is returned.

Note: Integer levels should be used when e.g. setting levels on instances of Logger and handlers. This function is used to convert between an integer level and the level name displayed in the formatted log output by means of the %(levelname)s format specifier (see LogRecord attributes).

logging.makeLogRecord(attrdict)
Creates and returns a new LogRecord instance whose attributes are defined by attrdict. This function is useful for taking a pickled LogRecord attribute dictionary, sent over a socket, and reconstituting it as a LogRecord instance at the receiving end.

logging.basicConfig([**kwargs])
Does basic configuration for the logging system by creating a StreamHandler with a default Formatter and adding it to the root logger. The functions debug(), info(), warning(), error() and critical() will call basicConfig() automatically if no handlers are defined for the root logger.

This function does nothing if the root logger already has handlers configured for it.

Changed in version 2.4: Formerly, basicConfig() did not take any keyword arguments.

Note: This function should be called from the main thread before other threads are started. In versions of Python prior to 2.7.1 and 3.2, if this function is called from multiple threads, it is possible

(in rare circumstances) that a handler will be added to the root logger more than once, leading to unexpected results such as messages being duplicated in the log.

The following keyword arguments are supported.

Format	Description
filename	Specifies that a FileHandler be created, using the specified filename, rather than a StreamHandler.
filemode	If filename is specified, open the file in this mode. Defaults to `'a'`.
format	Use the specified format string for the handler.
datefmt	Use the specified date/time format, as accepted by time.strftime().
level	Set the root logger level to the specified level.
stream	Use the specified stream to initialize the StreamHandler. Note that this argument is incompatible with filename - if both are present, stream is ignored.

logging.shutdown()
Informs the logging system to perform an orderly shutdown by flushing and closing all handlers. This should be called at application exit and no further use of the logging system should be made after this call.

logging.setLoggerClass(klass)
Tells the logging system to use the class klass when instantiating a logger. The class should define __init__() such that only a name argument is required, and the __init__() should call Logger.__init__(). This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior.

15.7.11 Integration with the warnings module

The captureWarnings() function can be used to integrate logging with the warnings module.

logging.captureWarnings(capture)
This function is used to turn the capture of warnings by logging on and off.

If capture is True, warnings issued by the warnings module will be redirected to the logging system. Specifically, a warning will be formatted using warnings.formatwarning() and the resulting string logged to a logger named `'py.warnings'` with a severity of WARNING.

If capture is False, the redirection of warnings to the logging system will stop, and warnings will be redirected to their original destinations (i.e. those in effect before captureWarnings(True) was called).

See also:

Module logging.config Configuration API for the logging module.

Module logging.handlers Useful handlers included with the logging module.

PEP 282 - A Logging System The proposal which described this feature for inclusion in the Python standard library.

Original Python logging package This is the original source for the logging package. The version of the package available from this site is suitable for use with Python 1.5.2, 2.1.x and 2.2.x, which do not include the logging package in the standard library.

15.8 logging.config — Logging configuration

Important

This page contains only reference information. For tutorials, please see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

Source code: Lib/logging/config.py

This section describes the API for configuring the logging module.

15.8.1 Configuration functions

The following functions configure the logging module. They are located in the logging.config module. Their use is optional — you can configure the logging module using these functions or by making calls to the main API (defined in logging itself) and defining handlers which are declared either in logging or logging.handlers.

logging.config.dictConfig(config)

Takes the logging configuration from a dictionary. The contents of this dictionary are described in Configuration dictionary schema below.

If an error is encountered during configuration, this function will raise a ValueError, TypeError, AttributeError or ImportError with a suitably descriptive message. The following is a (possibly incomplete) list of conditions which will raise an error:

- A level which is not a string or which is a string not corresponding to an actual logging level.
- A propagate value which is not a boolean.
- An id which does not have a corresponding destination.
- A non-existent handler id found during an incremental call.
- An invalid logger name.
- Inability to resolve to an internal or external object.

Parsing is performed by the DictConfigurator class, whose constructor is passed the dictionary used for configuration, and has a configure() method. The logging.config module has a callable attribute dictConfigClass which is initially set to DictConfigurator. You can replace the value of dictConfigClass with a suitable implementation of your own.

dictConfig() calls dictConfigClass passing the specified dictionary, and then calls the configure() method on the returned object to put the configuration into effect:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

For example, a subclass of DictConfigurator could call DictConfigurator.__init__() in its own __init__(), then set up custom prefixes which would be usable in the subsequent configure() call. dictConfigClass would be bound to this new subclass, and then dictConfig() could be called exactly as in the default, uncustomized state.

New in version 2.7.

logging.config.fileConfig(fname, defaults=None, disable_existing_loggers=True)

Reads the logging configuration from a configparser-format file named fname. The format of the file should be as described in Configuration file format. This function can be called several times from an application, allowing an end user to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration).

Parameters

- defaults – Defaults to be passed to the ConfigParser can be specified in this argument.
- disable_existing_loggers – If specified as False, loggers which exist when this call is made are left enabled. The default is True because this enables old behaviour in a backward-compatible way. This behaviour is to disable any existing loggers unless they or their ancestors are explicitly named in the logging configuration.

Changed in version 2.6: The disable_existing_loggers keyword argument was added. Previously, existing loggers were always disabled.

logging.config.listen(port=DEFAULT_LOGGING_CONFIG_PORT)

Starts up a socket server on the specified port, and listens for new configurations. If no port is specified, the module's default DEFAULT_LOGGING_CONFIG_PORT is used. Logging configurations will be sent as a file suitable for processing by fileConfig(). Returns a Thread instance on which you can call start() to start the server, and which you can join() when appropriate. To stop the server, call stopListening().

To send a configuration to the socket, read in the configuration file and send it to the socket as a string of bytes preceded by a four-byte length string packed in binary using struct.pack('>L', n).

Note: Because portions of the configuration are passed through eval(), use of this function may open its users to a security risk. While the function only binds to a socket on localhost, and so does not accept connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls listen(). Specifically, if the process calling listen() runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user's process, simply by connecting to the victim's listen() socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim's process. This is especially easy to do if the default port is used, but not hard even if a different port is used).

---

logging.config.stopListening()

Stops the listening server which was created with a call to listen(). This is typically called before calling join() on the return value from listen().

15.8.2 Configuration dictionary schema

Describing a logging configuration requires listing the various objects to create and the connections between them; for example, you may create a handler named 'console' and then say that the logger named 'startup' will send its messages to the 'console' handler. These objects aren't limited to those provided by the logging module because you might write your own formatter or handler class. The parameters to these classes

may also need to include external objects such as sys.stderr. The syntax for describing these objects and connections is defined in Object connections below.

Dictionary Schema Details

The dictionary passed to dictConfig() must contain the following keys:

- version - to be set to an integer value representing the schema version. The only valid value at present is 1, but having this key allows the schema to evolve while still preserving backwards compatibility.

All other keys are optional, but if present they will be interpreted as described below. In all cases below where a 'configuring dict' is mentioned, it will be checked for the special `'()'` key to see if a custom instantiation is required. If so, the mechanism described in User-defined objects below is used to create an instance; otherwise, the context is used to determine what to instantiate.

- formatters - the corresponding value will be a dict in which each key is a formatter id and each value is a dict describing how to configure the corresponding Formatter instance.

  The configuring dict is searched for keys format and datefmt (with defaults of None) and these are used to construct a Formatter instance.

- filters - the corresponding value will be a dict in which each key is a filter id and each value is a dict describing how to configure the corresponding Filter instance.

  The configuring dict is searched for the key name (defaulting to the empty string) and this is used to construct a logging.Filter instance.

- handlers - the corresponding value will be a dict in which each key is a handler id and each value is a dict describing how to configure the corresponding Handler instance.

  The configuring dict is searched for the following keys:

  - class (mandatory). This is the fully qualified name of the handler class.
  - level (optional). The level of the handler.
  - formatter (optional). The id of the formatter for this handler.
  - filters (optional). A list of ids of the filters for this handler.

  All other keys are passed through as keyword arguments to the handler's constructor. For example, given the snippet:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

  the handler with id console is instantiated as a logging.StreamHandler, using sys.stdout as the underlying stream. The handler with id file is instantiated as a logging.handlers.RotatingFileHandler with the keyword arguments filename='logconfig.log', maxBytes=1024, backupCount=3.

- loggers - the corresponding value will be a dict in which each key is a logger name and each value is a dict describing how to configure the corresponding Logger instance.

  The configuring dict is searched for the following keys:

  – level (optional). The level of the logger.

  – propagate (optional). The propagation setting of the logger.

  – filters (optional). A list of ids of the filters for this logger.

  – handlers (optional). A list of ids of the handlers for this logger.

  The specified loggers will be configured according to the level, propagation, filters and handlers specified.

- root - this will be the configuration for the root logger. Processing of the configuration will be as for any logger, except that the propagate setting will not be applicable.

- incremental - whether the configuration is to be interpreted as incremental to the existing configuration. This value defaults to False, which means that the specified configuration replaces the existing configuration with the same semantics as used by the existing fileConfig() API.

  If the specified value is True, the configuration is processed as described in the section on Incremental Configuration.

- disable_existing_loggers - whether any existing loggers are to be disabled. This setting mirrors the parameter of the same name in fileConfig(). If absent, this parameter defaults to True. This value is ignored if incremental is True.

Incremental Configuration

It is difficult to provide complete flexibility for incremental configuration. For example, because objects such as filters and formatters are anonymous, once a configuration is set up, it is not possible to refer to such anonymous objects when augmenting a configuration.

Furthermore, there is not a compelling case for arbitrarily altering the object graph of loggers, handlers, filters, formatters at run-time, once a configuration is set up; the verbosity of loggers and handlers can be controlled just by setting levels (and, in the case of loggers, propagation flags). Changing the object graph arbitrarily in a safe way is problematic in a multi-threaded environment; while not impossible, the benefits are not worth the complexity it adds to the implementation.

Thus, when the incremental key of a configuration dict is present and is True, the system will completely ignore any formatters and filters entries, and process only the level settings in the handlers entries, and the level and propagate settings in the loggers and root entries.

Using a value in the configuration dict lets configurations to be sent over the wire as pickled dicts to a socket listener. Thus, the logging verbosity of a long-running application can be altered over time with no need to stop and restart the application.

Object connections

The schema describes a set of logging objects - loggers, handlers, formatters, filters - which are connected to each other in an object graph. Thus, the schema needs to represent connections between the objects. For example, say that, once configured, a particular logger has attached to it a particular handler. For the purposes of this discussion, we can say that the logger represents the source, and the handler the destination, of a connection between the two. Of course in the configured objects this is represented by the logger holding a reference to the handler. In the configuration dict, this is done by giving each destination object an id

which identifies it unambiguously, and then using the id in the source object's configuration to indicate that a connection exists between the source and the destination object with that id.

So, for example, consider the following YAML snippet:

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

(Note: YAML used here because it's a little more readable than the equivalent Python source form for the dictionary.)

The ids for loggers are the logger names which would be used programmatically to obtain a reference to those loggers, e.g. foo.bar.baz. The ids for Formatters and Filters can be any string value (such as brief, precise above) and they are transient, in that they are only meaningful for processing the configuration dictionary and used to determine connections between objects, and are not persisted anywhere when the configuration call is complete.

The above snippet indicates that logger named foo.bar.baz should have two handlers attached to it, which are described by the handler ids h1 and h2. The formatter for h1 is that described by id brief, and the formatter for h2 is that described by id precise.

User-defined objects

The schema supports user-defined objects for handlers, filters and formatters. (Loggers do not need to have different types for different instances, so there is no support in this configuration schema for user-defined logger classes.)

Objects to be configured are described by dictionaries which detail their configuration. In some places, the logging system will be able to infer from the context how an object is to be instantiated, but when a user-defined object is to be instantiated, the system will not know how to do this. In order to provide complete flexibility for user-defined object instantiation, the user needs to provide a 'factory' - a callable which is called with a configuration dictionary and which returns the instantiated object. This is signalled by an absolute import path to the factory being made available under the special key `'()'`. Here's a concrete example:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
```

(continues on next page)

(continued from previous page)

```
    bar: baz
    spam: 99.9
    answer: 42
```

The above YAML snippet defines three formatters. The first, with id brief, is a standard logging.Formatter instance with the specified format string. The second, with id default, has a longer format and also defines the time format explicitly, and will result in a logging.Formatter initialized with those two format strings. Shown in Python source form, the brief and default formatters have configuration sub-dictionaries:

```
{
  'format' : '%(message)s'
}
```

and:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

respectively, and as these dictionaries do not contain the special key `'()'`, the instantiation is inferred from the context: as a result, standard logging.Formatter instances are created. The configuration sub-dictionary for the third formatter, with id custom, is:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

and this contains the special key `'()'`, which means that user-defined instantiation is wanted. In this case, the specified factory callable will be used. If it is an actual callable it will be used directly - otherwise, if you specify a string (as in the example) the actual callable will be located using normal import mechanisms. The callable will be called with the remaining items in the configuration sub-dictionary as keyword arguments. In the above example, the formatter with id custom will be assumed to be returned by the call:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

The key `'()'` has been used as the special key because it is not a valid keyword parameter name, and so will not clash with the names of the keyword arguments used in the call. The `'()'` also serves as a mnemonic that the corresponding value is a callable.

Access to external objects

There are times where a configuration needs to refer to objects external to the configuration, for example sys.stderr. If the configuration dict is constructed using Python code, this is straightforward, but a problem arises when the configuration is provided via a text file (e.g. JSON, YAML). In a text file, there is no standard way to distinguish sys.stderr from the literal string `'sys.stderr'`. To facilitate this distinction, the configuration system looks for certain special prefixes in string values and treat them specially. For example, if the literal string `'ext://sys.stderr'` is provided as a value in the configuration, then the ext:// will be stripped off and the remainder of the value processed using normal import mechanisms.

The handling of such prefixes is done in a way analogous to protocol handling: there is a generic mechanism to look for prefixes which match the regular expression ^(?P<prefix>[a-z]+)://(?P<suffix>.*)$ whereby, if

the prefix is recognised, the suffix is processed in a prefix-dependent manner and the result of the processing replaces the string value. If the prefix is not recognised, then the string value will be left as-is.

Access to internal objects

As well as external objects, there is sometimes also a need to refer to objects in the configuration. This will be done implicitly by the configuration system for things that it knows about. For example, the string value `'DEBUG'` for a level in a logger or handler will automatically be converted to the value logging.DEBUG, and the handlers, filters and formatter entries will take an object id and resolve to the appropriate destination object.

However, a more generic mechanism is needed for user-defined objects which are not known to the logging module. For example, consider logging.handlers.MemoryHandler, which takes a target argument which is another handler to delegate to. Since the system already knows about this class, then in the configuration, the given target just needs to be the object id of the relevant target handler, and the system will resolve to the handler from the id. If, however, a user defines a my.package.MyHandler which has an alternate handler, the configuration system would not know that the alternate referred to a handler. To cater for this, a generic resolution system allows the user to specify:

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

The literal string `'cfg://handlers.file'` will be resolved in an analogous way to strings with the ext:// prefix, but looking in the configuration itself rather than the import namespace. The mechanism allows access by dot or by index, in a similar way to that provided by str.format. Thus, given the following snippet:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

in the configuration, the string `'cfg://handlers'` would resolve to the dict with key handlers, the string `'cfg://handlers.email` would resolve to the dict with key email in the handlers dict, and so on. The string `'cfg://handlers.email.toaddrs[1]` would resolve to `'dev_team.domain.tld'` and the string `'cfg://handlers.email.toaddrs[0]'` would resolve to the value `'support_team@domain.tld'`. The subject value could be accessed using either `'cfg://handlers.email.subject'` or, equivalently, `'cfg://handlers.email[subject]'`. The latter form only needs to be used if the key contains spaces or non-alphanumeric characters. If an index value consists only of decimal digits, access will be attempted using the corresponding integer value, falling back to the string value if needed.

Given a string cfg://handlers.myhandler.mykey.123, this will resolve to `config_dict['handlers']['myhandler']['mykey']['123']`. If the string is specified as cfg://handlers.myhandler.mykey[123], the system will attempt to retrieve the value from `config_dict['handlers']['myhandler']['mykey'][123]`, and fall back to `config_dict['handlers']['myhandler']['mykey']['123']` if that fails.

Import resolution and custom importers

Import resolution, by default, uses the builtin __import__() function to do its importing. You may want to replace this with your own importing mechanism: if so, you can replace the importer attribute of the DictConfigurator or its superclass, the BaseConfigurator class. However, you need to be careful because of the way functions are accessed from classes via descriptors. If you are using a Python callable to do your imports, and you want to define it at class level rather than instance level, you need to wrap it with staticmethod(). For example:

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

You don't need to wrap with staticmethod() if you're setting the import callable on a configurator instance.

15.8.3 Configuration file format

The configuration file format understood by fileConfig() is based on configparser functionality. The file must contain sections called [loggers], [handlers] and [formatters] which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identifies how that entity is configured. Thus, for a logger named log01 in the [loggers] section, the relevant configuration details are held in a section [logger_log01]. Similarly, a handler called hand01 in the [handlers] section will have its configuration held in a section called [handler_hand01], while a formatter called form01 in the [formatters] section will have its configuration specified in a section called [formatter_form01]. The root logger configuration must be specified in a section called [logger_root].

Note: The fileConfig() API is older than the dictConfig() API and does not provide functionality to cover certain aspects of logging. For example, you cannot configure Filter objects, which provide for filtering of messages beyond simple integer levels, using fileConfig(). If you need to have instances of Filter in your logging configuration, you will need to use dictConfig(). Note that future enhancements to configuration functionality will be added to dictConfig(), so it's worth considering transitioning to this newer API when it's convenient to do so.

Examples of these sections in the file are given below.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```
[logger_root]
level=NOTSET
handlers=hand01
```

The level entry can be one of DEBUG, INFO, WARNING, ERROR, CRITICAL or NOTSET. For the root logger only, NOTSET means that all messages will be logged. Level values are eval()uated in the context of

the logging package's namespace.

The handlers entry is a comma-separated list of handler names, which must appear in the [handlers] section. These names must appear in the [handlers] section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

The level and handlers entries are interpreted as for the root logger, except that if a non-root logger's level is specified as NOTSET, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The propagate entry is set to 1 to indicate that messages must propagate to handlers higher up the logger hierarchy from this logger, or 0 to indicate that messages are not propagated to handlers up the hierarchy. The qualname entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.

Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The class entry indicates the handler's class (as determined by eval() in the logging package's namespace). The level is interpreted as for loggers, and NOTSET is taken to mean 'log everything'.

Changed in version 2.6: Added support for resolving the handler's class as a dotted module and class name.

The formatter entry indicates the key name of the formatter for this handler. If blank, a default formatter (logging._defaultFormatter) is used. If a name is specified, it must appear in the [formatters] section and have a corresponding section in the configuration file.

The args entry, when eval()uated in the context of the logging package's namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see how typical entries are constructed.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)
```

(continues on next page)

(continued from previous page)

```
[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
```

Sections which specify formatter configuration are typified by the following.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

The format entry is the overall format string, and the datefmt entry is the strftime()-compatible date/time format string. If empty, the package substitutes ISO8601 format date/times, which is almost equivalent to specifying the date format string `'%Y-%m-%d %H:%M:%S'`. The ISO8601 format also specifies milliseconds, which are appended to the result of using the above format string, with a comma separator. An example time in ISO8601 format is 2003-01-23 00:29:50,411.

The class entry is optional. It indicates the name of the formatter's class (as a dotted module and class name.) This option is useful for instantiating a Formatter subclass. Subclasses of Formatter can present exception tracebacks in an expanded or condensed format.

Note: Due to the use of eval() as described above, there are potential security risks which result from using the listen() to send and receive configurations via sockets. The risks are limited to where multiple users with no mutual trust run code on the same machine; see the listen() documentation for more information.

See also:

Module logging API reference for the logging module.

Module logging.handlers Useful handlers included with the logging module.

15.9 logging.handlers — Logging handlers

Important

This page contains only reference information. For tutorials, please see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

Source code: Lib/logging/handlers.py

The following useful handlers are provided in the package. Note that three of the handlers (StreamHandler, FileHandler and NullHandler) are actually defined in the logging module itself, but have been documented here along with the other handlers.

15.9.1 StreamHandler

The StreamHandler class, located in the core logging package, sends logging output to streams such as sys.stdout, sys.stderr or any file-like object (or, more precisely, any object which supports write() and flush() methods).

class logging.StreamHandler(stream=None)
Returns a new instance of the StreamHandler class. If stream is specified, the instance will use it for logging output; otherwise, sys.stderr will be used.

emit(record)
If a formatter is specified, it is used to format the record. The record is then written to the stream with a newline terminator. If exception information is present, it is formatted using traceback.print_exception() and appended to the stream.

flush()
Flushes the stream by calling its flush() method. Note that the close() method is inherited from Handler and so does no output, so an explicit flush() call may be needed at times.

15.9.2 FileHandler

The FileHandler class, located in the core logging package, sends logging output to a disk file. It inherits the output functionality from StreamHandler.

class logging.FileHandler(filename, mode='a', encoding=None, delay=False)
Returns a new instance of the FileHandler class. The specified file is opened and used as the stream for logging. If mode is not specified, `'a'` is used. If encoding is not None, it is used to open the file with that encoding. If delay is true, then file opening is deferred until the first call to emit(). By default, the file grows indefinitely.

Changed in version 2.6: delay was added.

close()
Closes the file.

emit(record)
Outputs the record to the file.

15.9.3 NullHandler

New in version 2.7.

The NullHandler class, located in the core logging package, does not do any formatting or output. It is essentially a 'no-op' handler for use by library developers.

class logging.NullHandler
Returns a new instance of the NullHandler class.

emit(record)
This method does nothing.

handle(record)
This method does nothing.

createLock()
This method returns None for the lock, since there is no underlying I/O to which access needs to be serialized.

See library-config for more information on how to use NullHandler.

15.9.4 WatchedFileHandler

New in version 2.6.

The WatchedFileHandler class, located in the logging.handlers module, is a FileHandler which watches the file it is logging to. If the file changes, it is closed and reopened using the file name.

A file change can happen because of usage of programs such as newsyslog and logrotate which perform log file rotation. This handler, intended for use under Unix/Linux, watches the file to see if it has changed since the last emit. (A file is deemed to have changed if its device or inode have changed.) If the file has changed, the old file stream is closed, and the file opened to get a new stream.

This handler is not appropriate for use under Windows, because under Windows open log files cannot be moved or renamed - logging opens the files with exclusive locks - and so there is no need for such a handler. Furthermore, ST_INO is not supported under Windows; stat() always returns zero for this value.

class logging.handlers.WatchedFileHandler(filename[, mode[, encoding[, delay]]])
Returns a new instance of the WatchedFileHandler class. The specified file is opened and used as the stream for logging. If mode is not specified, `'a'` is used. If encoding is not None, it is used to open the file with that encoding. If delay is true, then file opening is deferred until the first call to emit(). By default, the file grows indefinitely.

emit(record)
Outputs the record to the file, but first checks to see if the file has changed. If it has, the existing stream is flushed and closed and the file opened again, before outputting the record to the file.

15.9.5 RotatingFileHandler

The RotatingFileHandler class, located in the logging.handlers module, supports rotation of disk log files.

class logging.handlers.RotatingFileHandler(filename, mode='a', maxBytes=0, backupCount=0, encoding=None, delay=0)

Returns a new instance of the RotatingFileHandler class. The specified file is opened and used as the stream for logging. If mode is not specified, `'a'` is used. If encoding is not None, it is used to open the file with that encoding. If delay is true, then file opening is deferred until the first call to emit(). By default, the file grows indefinitely.

You can use the maxBytes and backupCount values to allow the file to rollover at a predetermined size. When the size is about to be exceeded, the file is closed and a new file is silently opened for output. Rollover occurs whenever the current log file is nearly maxBytes in length; if either of maxBytes or backupCount is zero, rollover never occurs. If backupCount is non-zero, the system will save old log files by appending the extensions '.1', '.2' etc., to the filename. For example, with a backupCount of 5 and a base file name of app.log, you would get app.log, app.log.1, app.log.2, up to app.log.5. The file being written to is always app.log. When this file is filled, it is closed and renamed to app.log.1, and if files app.log.1, app.log.2, etc. exist, then they are renamed to app.log.2, app.log.3 etc. respectively.

Changed in version 2.6: delay was added.

doRollover()

Does a rollover, as described above.

emit(record)

Outputs the record to the file, catering for rollover as described previously.

15.9.6 TimedRotatingFileHandler

The TimedRotatingFileHandler class, located in the logging.handlers module, supports rotation of disk log files at certain timed intervals.

class logging.handlers.TimedRotatingFileHandler(filename, when='h', interval=1, backupCount=0, encoding=None, delay=False, utc=False)

Returns a new instance of the TimedRotatingFileHandler class. The specified file is opened and used as the stream for logging. On rotating it also sets the filename suffix. Rotating happens based on the product of when and interval.

You can use the when to specify the type of interval. The list of possible values is below. Note that they are not case sensitive.

Value	Type of interval
`'S'`	Seconds
`'M'`	Minutes
`'H'`	Hours
`'D'`	Days
`'W0'-'W6'`	Weekday (0=Monday)
`'midnight'`	Roll over at midnight

When using weekday-based rotation, specify 'W0' for Monday, 'W1' for Tuesday, and so on up to 'W6' for Sunday. In this case, the value passed for interval isn't used.

The system will save old log files by appending extensions to the filename. The extensions are date-and-time based, using the strftime format %Y-%m-%d_%H-%M-%S or a leading portion thereof, depending on the rollover interval.

When computing the next rollover time for the first time (when the handler is created), the last modification time of an existing log file, or else the current time, is used to compute when the next rotation will occur.

If the utc argument is true, times in UTC will be used; otherwise local time is used.

If backupCount is nonzero, at most backupCount files will be kept, and if more would be created when rollover occurs, the oldest one is deleted. The deletion logic uses the interval to determine which files to delete, so changing the interval may leave old files lying around.

If delay is true, then file opening is deferred until the first call to emit().

Changed in version 2.6: delay and utc were added.

doRollover()
Does a rollover, as described above.

emit(record)
Outputs the record to the file, catering for rollover as described above.

15.9.7 SocketHandler

The SocketHandler class, located in the logging.handlers module, sends logging output to a network socket. The base class uses a TCP socket.

class logging.handlers.SocketHandler(host, port)
Returns a new instance of the SocketHandler class intended to communicate with a remote machine whose address is given by host and port.

close()
Closes the socket.

emit()
Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a LogRecord, use the makeLogRecord() function.

handleError()
Handles an error which has occurred during emit(). The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

makeSocket()
This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (socket.SOCK_STREAM).

makePickle(record)
Pickles the record's attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket.

Note that pickles aren't completely secure. If you are concerned about security, you may want to override this method to implement a more secure mechanism. For example, you can sign pickles using HMAC and then verify them on the receiving end, or alternatively you can disable unpickling of global objects on the receiving end.

send(packet)
Send a pickled string packet to the socket. This function allows for partial sends which can happen when the network is busy.

createSocket()
Tries to create a socket; on failure, uses an exponential back-off algorithm. On initial failure, the handler will drop the message it was trying to send. When subsequent messages are handled by the same instance, it will not try connecting until some time has passed. The default parameters

are such that the initial delay is one second, and if after that delay the connection still can't be made, the handler will double the delay each time up to a maximum of 30 seconds.

This behaviour is controlled by the following handler attributes:

- retryStart (initial delay, defaulting to 1.0 seconds).
- retryFactor (multiplier, defaulting to 2.0).
- retryMax (maximum delay, defaulting to 30.0 seconds).

This means that if the remote listener starts up after the handler has been used, you could lose messages (since the handler won't even attempt a connection until the delay has elapsed, but just silently drop messages during the delay period).

15.9.8 DatagramHandler

The DatagramHandler class, located in the logging.handlers module, inherits from SocketHandler to support sending logging messages over UDP sockets.

class logging.handlers.DatagramHandler(host, port)
Returns a new instance of the DatagramHandler class intended to communicate with a remote machine whose address is given by host and port.

emit()
Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a LogRecord, use the makeLogRecord() function.

makeSocket()
The factory method of SocketHandler is here overridden to create a UDP socket (socket.SOCK_DGRAM).

send(s)
Send a pickled string to a socket.

15.9.9 SysLogHandler

The SysLogHandler class, located in the logging.handlers module, supports sending logging messages to a remote or local Unix syslog.

class logging.handlers.SysLogHandler(address=('localhost', SYSLOG_UDP_PORT), facility=LOG_USER, socktype=socket.SOCK_DGRAM)
Returns a new instance of the SysLogHandler class intended to communicate with a remote Unix machine whose address is given by address in the form of a (host, port) tuple. If address is not specified, ('localhost', 514) is used. The address is used to open a socket. An alternative to providing a (host, port) tuple is providing an address as a string, for example '/dev/log'. In this case, a Unix domain socket is used to send the message to the syslog. If facility is not specified, LOG_USER is used. The type of socket opened depends on the socktype argument, which defaults to socket.SOCK_DGRAM and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of socket.SOCK_STREAM.

Note that if your server is not listening on UDP port 514, SysLogHandler may appear not to work. In that case, check what address you should be using for a domain socket - it's system dependent. For example, on Linux it's usually '/dev/log' but on OS/X it's '/var/run/syslog'. You'll need to check your platform and use the appropriate address (you may need to do this check at runtime if your application needs to run on several platforms). On Windows, you pretty much have to use the UDP option.

Changed in version 2.7: socktype was added.

close()
Closes the socket to the remote host.

emit(record)
The record is formatted, and then sent to the syslog server. If exception information is present, it is not sent to the server.

encodePriority(facility, priority)
Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.

The symbolic LOG_ values are defined in SysLogHandler and mirror the values defined in the sys/syslog.h header file.

Priorities

Name (string)	Symbolic value
alert	LOG_ALERT
crit or critical	LOG_CRIT
debug	LOG_DEBUG
emerg or panic	LOG_EMERG
err or error	LOG_ERR
info	LOG_INFO
notice	LOG_NOTICE
warn or warning	LOG_WARNING

Facilities

Name (string)	Symbolic value
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority(levelname)
Maps a logging level name to a syslog priority name. You may need to override this if you are using custom levels, or if the default algorithm is not suitable for your needs. The default algorithm

maps DEBUG, INFO, WARNING, ERROR and CRITICAL to the equivalent syslog names, and all other level names to 'warning'.

15.9.10 NTEventLogHandler

The NTEventLogHandler class, located in the logging.handlers module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond's Win32 extensions for Python installed.

class logging.handlers.NTEventLogHandler(appname, dllname=None, logtype='Application')

Returns a new instance of the NTEventLogHandler class. The appname is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The dllname should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, `'win32service.pyd'` is used - this is installed with the Win32 extensions and contains some basic placeholder message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The logtype is one of `'Application'`, `'System'` or `'Security'`, and defaults to `'Application'`.

close()

At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this.

emit(record)

Determines the message ID, event category and event type, and then logs the message in the NT event log.

getEventCategory(record)

Returns the event category for the record. Override this if you want to specify your own categories. This version returns 0.

getEventType(record)

Returns the event type for the record. Override this if you want to specify your own types. This version does a mapping using the handler's typemap attribute, which is set up in __init__() to a dictionary which contains mappings for DEBUG, INFO, WARNING, ERROR and CRITICAL. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler's typemap attribute.

getMessageID(record)

Returns the message ID for the record. If you are using your own messages, you could do this by having the msg passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base message ID in win32service.pyd.

15.9.11 SMTPHandler

The SMTPHandler class, located in the logging.handlers module, supports sending logging messages to an email address via SMTP.

class logging.handlers.SMTPHandler(mailhost, fromaddr, toaddrs, subject, credentials=None, secure=None)

Returns a new instance of the SMTPHandler class. The instance is initialized with the from and to

addresses and subject line of the email. The toaddrs should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the mailhost argument. If you use a string, the standard SMTP port is used. If your SMTP server requires authentication, you can specify a (username, password) tuple for the credentials argument.

To specify the use of a secure protocol (TLS), pass in a tuple to the secure argument. This will only be used when authentication credentials are supplied. The tuple should be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the smtplib.SMTP.starttls() method.)

Changed in version 2.6: credentials was added.

Changed in version 2.7: secure was added.

emit(record)
Formats the record and sends it to the specified addressees.

getSubject(record)
If you want to specify a subject line which is record-dependent, override this method.

15.9.12 MemoryHandler

The MemoryHandler class, located in the logging.handlers module, supports buffering of logging records in memory, periodically flushing them to a target handler. Flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

MemoryHandler is a subclass of the more general BufferingHandler, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling shouldFlush() to see if the buffer should be flushed. If it should, then flush() is expected to do the flushing.

class logging.handlers.BufferingHandler(capacity)
Initializes the handler with a buffer of the specified capacity.

emit(record)
Appends the record to the buffer. If shouldFlush() returns true, calls flush() to process the buffer.

flush()
You can override this to implement custom flushing behavior. This version just zaps the buffer to empty.

shouldFlush(record)
Returns true if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.

class logging.handlers.MemoryHandler(capacity, flushLevel=ERROR, target=None)
Returns a new instance of the MemoryHandler class. The instance is initialized with a buffer size of capacity. If flushLevel is not specified, ERROR is used. If no target is specified, the target will need to be set using setTarget() before this handler does anything useful.

close()
Calls flush(), sets the target to None and clears the buffer.

flush()
For a MemoryHandler, flushing means just sending the buffered records to the target, if there is one. The buffer is also cleared when this happens. Override if you want different behavior.

setTarget(target)
Sets the target handler for this handler.

shouldFlush(record)
Checks for buffer full or a record at the flushLevel or higher.

15.9.13 HTTPHandler

The HTTPHandler class, located in the logging.handlers module, supports sending logging messages to a Web server, using either GET or POST semantics.

class logging.handlers.HTTPHandler(host, url, method='GET')
Returns a new instance of the HTTPHandler class. The host can be of the form host:port, should you need to use a specific port number.

mapLogRecord(record)
Provides a dictionary, based on record, which is to be URL-encoded and sent to the web server. The default implementation just returns record.__dict__. This method can be overridden if e.g. only a subset of LogRecord is to be sent to the web server, or if more specific customization of what's sent to the server is required.

emit(record)
Sends the record to the Web server as a URL-encoded dictionary. The mapLogRecord() method is used to convert the record to the dictionary to be sent.

Note: Since preparing a record for sending it to a Web server is not the same as a generic formatting operation, using setFormatter() to specify a Formatter for a HTTPHandler has no effect. Instead of calling format(), this handler calls mapLogRecord() and then urllib.urlencode() to encode the dictionary in a form suitable for sending to a Web server.

See also:

Module logging API reference for the logging module.

Module logging.config Configuration API for the logging module.

15.10 getpass — Portable password input

The getpass module provides two functions:

getpass.getpass([prompt[, stream]])
Prompt the user for a password without echoing. The user is prompted using the string prompt, which defaults to 'Password: '. On Unix, the prompt is written to the file-like object stream. stream defaults to the controlling terminal (/dev/tty) or if that is unavailable to sys.stderr (this argument is ignored on Windows).

If echo free input is unavailable getpass() falls back to printing a warning message to stream and reading from sys.stdin and issuing a GetPassWarning.

Changed in version 2.5: The stream parameter was added.

Changed in version 2.6: On Unix it defaults to using /dev/tty before falling back to sys.stdin and sys.stderr.

Note: If you call getpass from within IDLE, the input may be done in the terminal you launched IDLE from rather than the idle window itself.

---

exception getpass.GetPassWarning
A UserWarning subclass issued when password input may be echoed.

getpass.getuser()
Return the "login name" of the user.

This function checks the environment variables LOGNAME, USER, LNAME and USERNAME, in order, and returns the value of the first one which is set to a non-empty string. If none are set, the login name from the password database is returned on systems which support the pwd module, otherwise, an exception is raised.

15.11 curses — Terminal handling for character-cell displays

Changed in version 1.6: Added support for the ncurses library and converted to a package.

The curses module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.

While curses is most widely used in the Unix environment, versions are available for DOS, OS/2, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of Unix.

Note: Since version 5.4, the ncurses library decides how to interpret non-ASCII data using the nl_langinfo function. That means that you have to call locale.setlocale() in the application and encode Unicode strings using one of the system's available encodings. This example uses the system's default encoding:

```
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

Then use code as the encoding for str.encode() calls.

See also:

Module curses.ascii Utilities for working with ASCII characters, regardless of your locale settings.

Module curses.panel A panel stack extension that adds depth to curses windows.

Module curses.textpad Editable text widget for curses supporting Emacs-like bindings.

curses-howto Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

The Demo/curses/ directory in the Python source distribution contains some example programs using the curses bindings provided by this module.

15.11.1 Functions

The module curses defines the following exception:

exception curses.error
Exception raised when a curses library function returns an error.

Note: Whenever x or y arguments to a function or a method are optional, they default to the current cursor location. Whenever attr is optional, it defaults to A_NORMAL.

The module curses defines the following functions:

curses.baudrate()
: Return the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

curses.beep()
: Emit a short attention sound.

curses.can_change_color()
: Return True or False, depending on whether the programmer can change the colors displayed by the terminal.

curses.cbreak()
: Enter cbreak mode. In cbreak mode (sometimes called "rare" mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first raw() then cbreak() leaves the terminal in cbreak mode.

curses.color_content(color_number)
: Return the intensity of the red, green, and blue (RGB) components in the color color_number, which must be between 0 and COLORS. A 3-tuple is returned, containing the R,G,B values for the given color, which will be between 0 (no component) and 1000 (maximum amount of component).

curses.color_pair(color_number)
: Return the attribute value for displaying text in the specified color. This attribute value can be combined with A_STANDOUT, A_REVERSE, and the other A_* attributes. pair_number() is the counterpart to this function.

curses.curs_set(visibility)
: Set the cursor state. visibility can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, the previous cursor state is returned; otherwise, an exception is raised. On many terminals, the "visible" mode is an underline cursor and the "very visible" mode is a block cursor.

curses.def_prog_mode()
: Save the current terminal mode as the "program" mode, the mode when the running program is using curses. (Its counterpart is the "shell" mode, for when the program is not in curses.) Subsequent calls to reset_prog_mode() will restore this mode.

curses.def_shell_mode()
: Save the current terminal mode as the "shell" mode, the mode when the running program is not using curses. (Its counterpart is the "program" mode, when the program is using curses capabilities.) Subsequent calls to reset_shell_mode() will restore this mode.

curses.delay_output(ms)
: Insert an ms millisecond pause in output.

curses.doupdate()
: Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The doupdate() ground updates the physical screen to match the virtual screen.

  The virtual screen may be updated by a noutrefresh() call after write operations such as addstr() have been performed on a window. The normal refresh() call is simply noutrefresh() followed by doupdate(); if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing noutrefresh() calls on all windows, followed by a single doupdate().

curses.echo()
Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

curses.endwin()
De-initialize the library, and return terminal to normal status.

curses.erasechar()
Return the user's current erase character. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.filter()
The filter() routine, if used, must be called before initscr() is called. The effect is that, during those calls, LINES is set to 1; the capabilities clear, cup, cud, cud1, cuu1, cuu, vpa are disabled; and the home string is set to the value of cr. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

curses.flash()
Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as 'visible bell' to the audible attention signal produced by beep().

curses.flushinp()
Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

curses.getmouse()
After getch() returns KEY_MOUSE to signal a mouse event, this method should be call to retrieve the queued mouse event, represented as a 5-tuple (id, x, y, z, bstate). id is an ID value used to distinguish multiple devices, and x, y, z are the event's coordinates. (z is currently unused.) bstate is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where n is the button number from 1 to 4: BUTTONn_PRESSED, BUTTONn_RELEASED, BUTTONn_CLICKED, BUTTONn_DOUBLE_CLICKED, BUTTONn_TRIPLE_CLICKED, BUTTON_SHIFT, BUTTON_CTRL, BUTTON_ALT.

curses.getsyx()
Return the current coordinates of the virtual screen cursor in y and x. If leaveok is currently true, then -1,-1 is returned.

curses.getwin(file)
Read window related data stored in the file by an earlier putwin() call. The routine then creates and initializes a new window using that data, returning the new window object.

curses.has_colors()
Return True if the terminal can display colors; otherwise, return False.

curses.has_ic()
Return True if the terminal has insert- and delete-character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.has_il()
Return True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.has_key(ch)
Take a key value ch, and return True if the current terminal type recognizes a key with that value.

curses.halfdelay(tenths)
Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are im-

mediately available to the program. However, after blocking for tenths tenths of seconds, an exception is raised if nothing has been typed. The value of tenths must be a number between 1 and 255. Use nocbreak() to leave half-delay mode.

curses.init_color(color_number, r, g, b)
Change the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of color_number must be between 0 and COLORS. Each of r, g, b, must be a value between 0 and 1000. When init_color() is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if can_change_color() returns 1.

curses.init_pair(pair_number, fg, bg)
Change the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of pair_number must be between 1 and COLOR_PAIRS - 1 (the 0 color pair is wired to white on black and cannot be changed). The value of fg and bg arguments must be between 0 and COLORS. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

curses.initscr()
Initialize the library. Return a WindowObject which represents the whole screen.

Note: If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

---

curses.is_term_resized(nlines, ncols)
Return True if resize_term() would modify the window structure, False otherwise.

curses.isendwin()
Return True if endwin() has been called (that is, the curses library has been deinitialized).

curses.keyname(k)
Return the name of the key numbered k. The name of a key generating printable ASCII character is the key's character. The name of a control-key combination is a two-character string consisting of a caret followed by the corresponding printable ASCII character. The name of an alt-key combination (128–255) is a string consisting of the prefix 'M-' followed by the name of the corresponding ASCII character.

curses.killchar()
Return the user's current line kill character. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.longname()
Return a string containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to initscr().

curses.meta(yes)
If yes is 1, allow 8-bit characters to be input. If yes is 0, allow only 7-bit chars.

curses.mouseinterval(interval)
Set the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and return the previous interval value. The default value is 200 msec, or one fifth of a second.

curses.mousemask(mousemask)
Set the mouse events to be reported, and return a tuple (availmask, oldmask). availmask indicates which of the specified mouse events can be reported; on complete failure it returns 0. oldmask is the

previous value of the given window's mouse event mask. If this function is never called, no mouse events are ever reported.

curses.napms(ms)
Sleep for ms milliseconds.

curses.newpad(nlines, ncols)
Create and return a pointer to a new pad data structure with the given number of lines and columns. A pad is returned as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The refresh() and noutrefresh() methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol; the p arguments refer to the upper left corner of the pad region to be displayed and the s arguments define a clipping box on the screen within which the pad region is to be displayed.

curses.newwin(nlines, ncols)
curses.newwin(nlines, ncols, begin_y, begin_x)
Return a new window, whose left-upper corner is at (begin_y, begin_x), and whose height/width is nlines/ncols.

By default, the window will extend from the specified position to the lower right corner of the screen.

curses.nl()
Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

curses.nocbreak()
Leave cbreak mode. Return to normal "cooked" mode with line buffering.

curses.noecho()
Leave echo mode. Echoing of input characters is turned off.

curses.nonl()
Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of addch('\n'), which always does the equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

curses.noqiflush()
When the noqiflush() routine is used, normal flush of input and output queues associated with the INTR, QUIT and SUSP characters will not be done. You may want to call noqiflush() in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

curses.noraw()
Leave raw mode. Return to normal "cooked" mode with line buffering.

curses.pair_content(pair_number)
Return a tuple (fg, bg) containing the colors for the requested color pair. The value of pair_number must be between 1 and COLOR_PAIRS - 1.

curses.pair_number(attr)
Return the number of the color-pair set by the attribute value attr. color_pair() is the counterpart to this function.

curses.putp(string)
: Equivalent to tputs(str, 1, putchar); emit the value of a specified terminfo capability for the current terminal. Note that the output of putp() always goes to standard output.

curses.qiflush([flag])
: If flag is False, the effect is the same as calling noqiflush(). If flag is True, or no argument is provided, the queues will be flushed when these control characters are read.

curses.raw()
: Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

curses.reset_prog_mode()
: Restore the terminal to "program" mode, as previously saved by def_prog_mode().

curses.reset_shell_mode()
: Restore the terminal to "shell" mode, as previously saved by def_shell_mode().

curses.resetty()
: Restore the state of the terminal modes to what it was at the last call to savetty().

curses.resize_term(nlines, ncols)
: Backend function used by resizeterm(), performing most of the work; when resizing the windows, resize_term() blank-fills the areas that are extended. The calling application should fill in these areas with appropriate data. The resize_term() function attempts to resize all windows. However, due to the calling convention of pads, it is not possible to resize these without additional interaction with the application.

curses.resizeterm(nlines, ncols)
: Resize the standard and current windows to the specified dimensions, and adjusts other bookkeeping data used by the curses library that record the window dimensions (in particular the SIGWINCH handler).

curses.savetty()
: Save the current state of the terminal modes in a buffer, usable by resetty().

curses.setsyx(y, x)
: Set the virtual screen cursor to y, x. If y and x are both -1, then leaveok is set.

curses.setupterm([termstr, fd])
: Initialize the terminal. termstr is a string giving the terminal name; if omitted, the value of the TERM environment variable will be used. fd is the file descriptor to which any initialization sequences will be sent; if not supplied, the file descriptor for sys.stdout will be used.

curses.start_color()
: Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after initscr().

  start_color() initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the curses module, COLORS and COLOR_PAIRS, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

curses.termattrs()
: Return a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

curses.termname()
: Return the value of the environment variable TERM, truncated to 14 characters.

curses.tigetflag(capname)
Return the value of the Boolean capability corresponding to the terminfo capability name capname. The value -1 is returned if capname is not a Boolean capability, or 0 if it is canceled or absent from the terminal description.

curses.tigetnum(capname)
Return the value of the numeric capability corresponding to the terminfo capability name capname. The value -2 is returned if capname is not a numeric capability, or -1 if it is canceled or absent from the terminal description.

curses.tigetstr(capname)
Return the value of the string capability corresponding to the terminfo capability name capname. None is returned if capname is not a string capability, or is canceled or absent from the terminal description.

curses.tparm(str[, ...])
Instantiate the string str with the supplied parameters, where str should be a parameterized string obtained from the terminfo database. E.g. tparm(tigetstr("cup"), 5, 3) could result in '\033[6;4H', the exact result depending on terminal type.

curses.typeahead(fd)
Specify that the file descriptor fd be used for typeahead checking. If fd is -1, then no typeahead checking is done.

The curses library does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or doupdate is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

curses.unctrl(ch)
Return a string which is a printable representation of the character ch. Control characters are displayed as a caret followed by the character, for example as ^C. Printing characters are left as they are.

curses.ungetch(ch)
Push ch so the next getch() will return it.

Note: Only one ch can be pushed before getch() is called.

---

curses.ungetmouse(id, x, y, z, bstate)
Push a KEY_MOUSE event onto the input queue, associating the given state data with it.

curses.use_env(flag)
If used, this function should be called before initscr() or newterm are called. When flag is False, the values of lines and columns specified in the terminfo database will be used, even if environment variables LINES and COLUMNS (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if LINES and COLUMNS are not set).

curses.use_default_colors()
Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application. The default color is assigned to the color number -1. After calling this function, init_pair(x, curses.COLOR_RED, -1) initializes, for instance, color pair x to a red foreground color on the default background.

curses.wrapper(func, ...)
Initialize curses and call another callable object, func, which should be the rest of your curses-using application. If the application raises an exception, this function will restore the terminal to a sane state before re-raising the exception and generating a traceback. The callable object func is then passed the main window 'stdscr' as its first argument, followed by any other arguments passed to wrapper(). Before calling func, wrapper() turns on cbreak mode, turns off echo, enables the terminal keypad,

and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.

15.11.2 Window Objects

Window objects, as returned by initscr() and newwin() above, have the following methods:

window.addch(ch[, attr])
window.addch(y, x, ch[, attr])

Note: A character means a C character (an ASCII code), rather than a Python character (a string of length 1). (This note is true whenever the documentation mentions a character.) The built-in ord() is handy for conveying strings to codes.

Paint character ch at (y, x) with attributes attr, overwriting any character previously painter at that location. By default, the character position and attributes are the current settings for the window object.

Note: Writing outside the window, subwindow, or pad raises a curses.error. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the character is printed.

---

window.addnstr(str, n[, attr])
window.addnstr(y, x, str, n[, attr])
Paint at most n characters of the string str at (y, x) with attributes attr, overwriting anything previously on the display.

window.addstr(str[, attr])
window.addstr(y, x, str[, attr])
Paint the string str at (y, x) with attributes attr, overwriting anything previously on the display.

Note: Writing outside the window, subwindow, or pad raises curses.error. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the string is printed.

---

window.attroff(attr)
Remove attribute attr from the "background" set applied to all writes to the current window.

window.attron(attr)
Add attribute attr from the "background" set applied to all writes to the current window.

window.attrset(attr)
Set the "background" set of attributes to attr. This set is initially 0 (no attributes).

window.bkgd(ch[, attr])
Set the background property of the window to the character ch, with attributes attr. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.

- Wherever the former background character appears, it is changed to the new background character.

window.bkgdset(ch[, attr])
Set the window's background. A window's background consists of a character and any combination of attributes. The attribute part of the background is combined (OR'ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

window.border([ls[, rs[, ts[, bs[, tl[, tr[, bl[, br]]]]]]]])
Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details. The characters can be specified as integers or as one-character strings.

Note: A 0 value for any parameter will cause the default character to be used for that parameter. Keyword parameters can not be used. The defaults are listed in this table:

Parameter	Description	Default value
ls	Left side	ACS_VLINE
rs	Right side	ACS_VLINE
ts	Top	ACS_HLINE
bs	Bottom	ACS_HLINE
tl	Upper-left corner	ACS_ULCORNER
tr	Upper-right corner	ACS_URCORNER
bl	Bottom-left corner	ACS_LLCORNER
br	Bottom-right corner	ACS_LRCORNER

window.box([vertch, horch])
Similar to border(), but both ls and rs are vertch and both ts and bs are horch. The default corner characters are always used by this function.

window.chgat(attr)
window.chgat(num, attr)
window.chgat(y, x, attr)
window.chgat(y, x, num, attr)
Set the attributes of num characters at the current cursor position, or at position (y, x) if supplied. If num is not given or is -1, the attribute will be set on all the characters to the end of the line. This function moves cursor to position (y, x) if supplied. The changed line will be touched using the touchline() method so that the contents will be redisplayed by the next window refresh.

window.clear()
Like erase(), but also cause the whole window to be repainted upon next call to refresh().

window.clearok(yes)
If yes is 1, the next call to refresh() will clear the window completely.

window.clrtobot()
Erase from cursor to the end of the window: all lines below the cursor are deleted, and then the equivalent of clrtoeol() is performed.

window.clrtoeol()
Erase from cursor to the end of the line.

window.cursyncup()

Update the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

window.delch([y, x])
Delete any character at (y, x).

window.deleteln()
Delete the line under the cursor. All following lines are moved up by one line.

window.derwin(begin_y, begin_x)
window.derwin(nlines, ncols, begin_y, begin_x)
An abbreviation for "derive window", derwin() is the same as calling subwin(), except that begin_y and begin_x are relative to the origin of the window, rather than relative to the entire screen. Return a window object for the derived window.

window.echochar(ch[, attr])
Add character ch with attribute attr, and immediately call refresh() on the window.

window.enclose(y, x)
Test whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning True or False. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

window.erase()
Clear the window.

window.getbegyx()
Return a tuple (y, x) of co-ordinates of upper-left corner.

window.getbkgd()
Return the given window's current background character/attribute pair.

window.getch([y, x])
Get a character. Note that the integer returned does not have to be in ASCII range: function keys, keypad keys and so on return numbers higher than 256. In no-delay mode, -1 is returned if there is no input, else getch() waits until a key is pressed.

window.getkey([y, x])
Get a character, returning a string instead of an integer, as getch() does. Function keys, keypad keys and so on return a multibyte string containing the key name. In no-delay mode, an exception is raised if there is no input.

window.getmaxyx()
Return a tuple (y, x) of the height and width of the window.

window.getparyx()
Return the beginning coordinates of this window relative to its parent window into two integer variables y and x. Return -1, -1 if this window has no parent.

window.getstr([y, x])
Read a string from the user, with primitive line editing capacity.

window.getyx()
Return a tuple (y, x) of current cursor position relative to the window's upper-left corner.

window.hline(ch, n)
window.hline(y, x, ch, n)
Display a horizontal line starting at (y, x) with length n consisting of the character ch.

window.idcok(flag)
If flag is False, curses no longer considers using the hardware insert/delete character feature of the

terminal; if flag is True, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

window.idlok(yes)
: If called with yes equal to 1, curses will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

window.immedok(flag)
: If flag is True, any change in the window image automatically causes the window to be refreshed; you no longer have to call refresh() yourself. However, it may degrade performance considerably, due to repeated calls to wrefresh. This option is disabled by default.

window.inch([y, x])
: Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

window.insch(ch[, attr])
window.insch(y, x, ch[, attr])
: Paint character ch at (y, x) with attributes attr, moving the line from position x right by one character.

window.insdelln(nlines)
: Insert nlines lines into the specified window above the current line. The nlines bottom lines are lost. For negative nlines, delete nlines lines starting with the one under the cursor, and move the remaining lines up. The bottom nlines lines are cleared. The current cursor position remains the same.

window.insertln()
: Insert a blank line under the cursor. All following lines are moved down by one line.

window.insnstr(str, n[, attr])
window.insnstr(y, x, str, n[, attr])
: Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to n characters. If n is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to y, x, if specified).

window.insstr(str[, attr])
window.insstr(y, x, str[, attr])
: Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to y, x, if specified).

window.instr([n])
window.instr(y, x[, n])
: Return a string of characters, extracted from the window starting at the current cursor position, or at y, x if specified. Attributes are stripped from the characters. If n is specified, instr() returns a string at most n characters long (exclusive of the trailing NUL).

window.is_linetouched(line)
: Return True if the specified line was modified since the last call to refresh(); otherwise return False. Raise a curses.error exception if line is not valid for the given window.

window.is_wintouched()
: Return True if the specified window was modified since the last call to refresh(); otherwise return False.

window.keypad(yes)
: If yes is 1, escape sequences generated by some keys (keypad, function keys) will be interpreted by curses. If yes is 0, escape sequences will be left as is in the input stream.

window.leaveok(yes)
: If yes is 1, cursor is left where it is on update, instead of being at "cursor position." This reduces cursor movement where possible. If possible the cursor will be made invisible.

  If yes is 0, cursor will always be at "cursor position" after an update.

window.move(new_y, new_x)
: Move cursor to (new_y, new_x).

window.mvderwin(y, x)
: Move the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

window.mvwin(new_y, new_x)
: Move the window so its upper-left corner is at (new_y, new_x).

window.nodelay(yes)
: If yes is 1, getch() will be non-blocking.

window.notimeout(yes)
: If yes is 1, escape sequences will not be timed out.

  If yes is 0, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

window.noutrefresh()
: Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call doupdate().

window.overlay(destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])
: Overlay the window on top of destwin. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of destwin.

  To get fine-grained control over the copied region, the second form of overlay() can be used. sminrow and smincol are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

window.overwrite(destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])
: Overwrite the window on top of destwin. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of destwin.

  To get fine-grained control over the copied region, the second form of overwrite() can be used. sminrow and smincol are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

window.putwin(file)
: Write all data associated with the window into the provided file object. This information can be later retrieved using the getwin() function.

window.redrawln(beg, num)
: Indicate that the num screen lines, starting at line beg, are corrupted and should be completely redrawn on the next refresh() call.

window.redrawwin()
: Touch the entire window, causing it to be completely redrawn on the next refresh() call.

window.refresh([pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol])
: Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with newpad(). The additional parameters are needed to indicate what part of the pad and screen are involved. pminrow and pmincol specify the upper left-hand corner of the rectangle to be displayed in the pad. sminrow, smincol, smaxrow, and smaxcol specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of pminrow, pmincol, sminrow, or smincol are treated as if they were zero.

window.resize(nlines, ncols)
Reallocate storage for a curses window to adjust its dimensions to the specified values. If either dimension is larger than the current values, the window's data is filled with blanks that have the current background rendition (as set by bkgdset()) merged into them.

window.scroll([lines=1])
Scroll the screen or scrolling region upward by lines lines.

window.scrollok(flag)
Control what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If flag is false, the cursor is left on the bottom line. If flag is true, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call idlok().

window.setscrreg(top, bottom)
Set the scrolling region from line top to line bottom. All scrolling actions will take place in this region.

window.standend()
Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

window.standout()
Turn on attribute A_STANDOUT.

window.subpad(begin_y, begin_x)
window.subpad(nlines, ncols, begin_y, begin_x)
Return a sub-window, whose upper-left corner is at (begin_y, begin_x), and whose width/height is ncols/nlines.

window.subwin(begin_y, begin_x)
window.subwin(nlines, ncols, begin_y, begin_x)
Return a sub-window, whose upper-left corner is at (begin_y, begin_x), and whose width/height is ncols/nlines.

By default, the sub-window will extend from the specified position to the lower right corner of the window.

window.syncdown()
Touch each location in the window that has been touched in any of its ancestor windows. This routine is called by refresh(), so it should almost never be necessary to call it manually.

window.syncok(flag)
If called with flag set to True, then syncup() is called automatically whenever there is a change in the window.

window.syncup()
Touch all locations in ancestors of the window that have been changed in the window.

window.timeout(delay)
Set blocking or non-blocking read behavior for the window. If delay is negative, blocking read is used (which will wait indefinitely for input). If delay is zero, then non-blocking read is used, and -1 will

be returned by getch() if no input is waiting. If delay is positive, then getch() will block for delay milliseconds, and return -1 if there is still no input at the end of that time.

window.touchline(start, count[, changed])
Pretend count lines have been changed, starting with line start. If changed is supplied, it specifies whether the affected lines are marked as having been changed (changed=1) or unchanged (changed=0).

window.touchwin()
Pretend the whole window has been changed, for purposes of drawing optimizations.

window.untouchwin()
Mark all lines in the window as unchanged since the last call to refresh().

window.vline(ch, n)
window.vline(y, x, ch, n)
Display a vertical line starting at (y, x) with length n consisting of the character ch.

15.11.3 Constants

The curses module defines the following data members:

curses.ERR
Some curses routines that return an integer, such as getch(), return ERR upon failure.

curses.OK
Some curses routines that return an integer, such as napms(), return OK upon success.

curses.version
A string representing the current version of the module. Also available as __version__.

Some constants are available to specify character cell attributes. The exact constants available are system dependent.

Attribute	Meaning
A_ALTCHARSET	Alternate character set mode
A_BLINK	Blink mode
A_BOLD	Bold mode
A_DIM	Dim mode
A_INVIS	Invisible or blank mode
A_NORMAL	Normal attribute
A_PROTECT	Protected mode
A_REVERSE	Reverse background and foreground colors
A_STANDOUT	Standout mode
A_UNDERLINE	Underline mode
A_HORIZONTAL	Horizontal highlight
A_LEFT	Left highlight
A_LOW	Low highlight
A_RIGHT	Right highlight
A_TOP	Top highlight
A_VERTICAL	Vertical highlight
A_CHARTEXT	Bit-mask to extract a character

Several constants are available to extract corresponding attributes returned by some methods.

Bit-mask	Meaning
A_ATTRIBUTES	Bit-mask to extract attributes
A_CHARTEXT	Bit-mask to extract a character
A_COLOR	Bit-mask to extract color-pair field information

Keys are referred to by integer constants with names starting with KEY_. The exact keycaps available are system dependent.

Key constant	Key
KEY_MIN	Minimum key value
KEY_BREAK	Break key (unreliable)
KEY_DOWN	Down-arrow
KEY_UP	Up-arrow
KEY_LEFT	Left-arrow
KEY_RIGHT	Right-arrow
KEY_HOME	Home key (upward+left arrow)
KEY_BACKSPACE	Backspace (unreliable)
KEY_F0	Function keys. Up to 64 function keys are supported.
KEY_Fn	Value of function key n
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_EIC	Exit insert char mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send (unreliable)
KEY_SRESET	Soft (partial) reset (unreliable)
KEY_RESET	Reset or hard reset (unreliable)
KEY_PRINT	Print
KEY_LL	Home down or bottom (lower left)
KEY_A1	Upper left of keypad
KEY_A3	Upper right of keypad
KEY_B2	Center of keypad
KEY_C1	Lower left of keypad
KEY_C3	Lower right of keypad
KEY_BTAB	Back tab
KEY_BEG	Beg (beginning)
KEY_CANCEL	Cancel
KEY_CLOSE	Close
KEY_COMMAND	Cmd (command)
KEY_COPY	Copy

Continued on next page

Table 1 – continued from previous page

Key constant	Key
KEY_CREATE	Create
KEY_END	End
KEY_EXIT	Exit
KEY_FIND	Find
KEY_HELP	Help
KEY_MARK	Mark
KEY_MESSAGE	Message
KEY_MOVE	Move
KEY_NEXT	Next
KEY_OPEN	Open
KEY_OPTIONS	Options
KEY_PREVIOUS	Prev (previous)
KEY_REDO	Redo
KEY_REFERENCE	Ref (reference)
KEY_REFRESH	Refresh
KEY_REPLACE	Replace
KEY_RESTART	Restart
KEY_RESUME	Resume
KEY_SAVE	Save
KEY_SBEG	Shifted Beg (beginning)
KEY_SCANCEL	Shifted Cancel
KEY_SCOMMAND	Shifted Command
KEY_SCOPY	Shifted Copy
KEY_SCREATE	Shifted Create
KEY_SDC	Shifted Delete char
KEY_SDL	Shifted Delete line
KEY_SELECT	Select
KEY_SEND	Shifted End
KEY_SEOL	Shifted Clear line
KEY_SEXIT	Shifted Exit
KEY_SFIND	Shifted Find
KEY_SHELP	Shifted Help
KEY_SHOME	Shifted Home
KEY_SIC	Shifted Input
KEY_SLEFT	Shifted Left arrow
KEY_SMESSAGE	Shifted Message
KEY_SMOVE	Shifted Move
KEY_SNEXT	Shifted Next
KEY_SOPTIONS	Shifted Options
KEY_SPREVIOUS	Shifted Prev
KEY_SPRINT	Shifted Print
KEY_SREDO	Shifted Redo
KEY_SREPLACE	Shifted Replace
KEY_SRIGHT	Shifted Right arrow
KEY_SRSUME	Shifted Resume
KEY_SSAVE	Shifted Save
KEY_SSUSPEND	Shifted Suspend
KEY_SUNDO	Shifted Undo
KEY_SUSPEND	Suspend

Continued on next page

Table 1 – continued from previous page

Key constant	Key
KEY_UNDO	Undo
KEY_MOUSE	Mouse event has occurred
KEY_RESIZE	Terminal resize event
KEY_MAX	Maximum key value

On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (KEY_F1, KEY_F2, KEY_F3, KEY_F4) available, and the arrow keys mapped to KEY_UP, KEY_DOWN, KEY_LEFT and KEY_RIGHT in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard:

Keycap	Constant
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

Note: These are available only after initscr() has been called.

ACS code	Meaning
ACS_BBSS	alternate name for upper right corner
ACS_BLOCK	solid square block
ACS_BOARD	board of squares
ACS_BSBS	alternate name for horizontal line
ACS_BSSB	alternate name for upper left corner
ACS_BSSS	alternate name for top tee
ACS_BTEE	bottom tee
ACS_BULLET	bullet
ACS_CKBOARD	checker board (stipple)
ACS_DARROW	arrow pointing down
ACS_DEGREE	degree symbol
ACS_DIAMOND	diamond
ACS_GEQUAL	greater-than-or-equal-to
ACS_HLINE	horizontal line
ACS_LANTERN	lantern symbol
ACS_LARROW	left arrow
ACS_LEQUAL	less-than-or-equal-to
ACS_LLCORNER	lower left-hand corner
ACS_LRCORNER	lower right-hand corner
ACS_LTEE	left tee
ACS_NEQUAL	not-equal sign

Continued on next page

Table 2 – continued from previous page

ACS code	Meaning
ACS_PI	letter pi
ACS_PLMINUS	plus-or-minus sign
ACS_PLUS	big plus sign
ACS_RARROW	right arrow
ACS_RTEE	right tee
ACS_S1	scan line 1
ACS_S3	scan line 3
ACS_S7	scan line 7
ACS_S9	scan line 9
ACS_SBBS	alternate name for lower right corner
ACS_SBSB	alternate name for vertical line
ACS_SBSS	alternate name for right tee
ACS_SSBB	alternate name for lower left corner
ACS_SSBS	alternate name for bottom tee
ACS_SSSB	alternate name for left tee
ACS_SSSS	alternate name for crossover or big plus
ACS_STERLING	pound sterling
ACS_TTEE	top tee
ACS_UARROW	up arrow
ACS_ULCORNER	upper left corner
ACS_URCORNER	upper right corner
ACS_VLINE	vertical line

The following table lists the predefined colors:

Constant	Color
COLOR_BLACK	Black
COLOR_BLUE	Blue
COLOR_CYAN	Cyan (light greenish blue)
COLOR_GREEN	Green
COLOR_MAGENTA	Magenta (purplish red)
COLOR_RED	Red
COLOR_WHITE	White
COLOR_YELLOW	Yellow

15.12 curses.textpad — Text input widget for curses programs

New in version 1.6.

The curses.textpad module provides a Textbox class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module curses.textpad defines the following function:

curses.textpad.rectangle(win, uly, ulx, lry, lrx)

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x

coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

15.12.1 Textbox objects

You can instantiate a Textbox object as follows:

class curses.textpad.Textbox(win)
Return a textbox widget object. The win argument should be a curses WindowObject in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates (0, 0). The instance's stripspaces flag is initially on.

Textbox objects have the following methods:

edit([validator])
This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If validator is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the stripspaces attribute.

do_command(ch)
Process a single command keystroke. Here are the supported special keystrokes:

Keystroke	Action
Control-A	Go to left edge of window.
Control-B	Cursor left, wrapping to previous line if appropriate.
Control-D	Delete character under cursor.
Control-E	Go to right edge (stripspaces off) or end of line (stripspaces on).
Control-F	Cursor right, wrapping to next line when appropriate.
Control-G	Terminate, returning the window contents.
Control-H	Delete character backward.
Control-J	Terminate if the window is 1 line, otherwise insert newline.
Control-K	If line is blank, delete it, otherwise clear to end of line.
Control-L	Refresh screen.
Control-N	Cursor down; move down one line.
Control-O	Insert a blank line at cursor location.
Control-P	Cursor up; move up one line.

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible:

Constant	Keystroke
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

gather()
: Return the window contents as a string; whether blanks in the window are included is affected by the stripspaces member.

stripspaces
: This attribute is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.

15.13 curses.ascii — Utilities for ASCII characters

New in version 1.6.

The curses.ascii module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows:

Name	Meaning
NUL	
SOH	Start of heading, console interrupt
STX	Start of text
ETX	End of text
EOT	End of transmission
ENQ	Enquiry, goes with ACK flow control
ACK	Acknowledgement
BEL	Bell
BS	Backspace
TAB	Tab
HT	Alias for TAB: “Horizontal tab”
LF	Line feed
NL	Alias for LF: “New line”
VT	Vertical tab
FF	Form feed
CR	Carriage return
SO	Shift-out, begin alternate character set
SI	Shift-in, resume default character set
DLE	Data-link escape
DC1	XON, for flow control
DC2	Device control 2, block-mode flow control
DC3	XOFF, for flow control
DC4	Device control 4
NAK	Negative acknowledgement
SYN	Synchronous idle
ETB	End transmission block
CAN	Cancel
EM	End of medium
SUB	Substitute
ESC	Escape
FS	File separator
GS	Group separator
RS	Record separator, block-mode terminator

Continued on next page

Table 3 – continued from previous page

Name	Meaning
US	Unit separator
SP	Space
DEL	Delete

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library:

curses.ascii.isalnum(c)
Checks for an ASCII alphanumeric character; it is equivalent to isalpha(c) or isdigit(c).

curses.ascii.isalpha(c)
Checks for an ASCII alphabetic character; it is equivalent to isupper(c) or islower(c).

curses.ascii.isascii(c)
Checks for a character value that fits in the 7-bit ASCII set.

curses.ascii.isblank(c)
Checks for an ASCII whitespace character; space or horizontal tab.

curses.ascii.iscntrl(c)
Checks for an ASCII control character (in the range 0x00 to 0x1f or 0x7f).

curses.ascii.isdigit(c)
Checks for an ASCII decimal digit, `'0'` through `'9'`. This is equivalent to c in string.digits.

curses.ascii.isgraph(c)
Checks for ASCII any printable character except space.

curses.ascii.islower(c)
Checks for an ASCII lower-case character.

curses.ascii.isprint(c)
Checks for any ASCII printable character including space.

curses.ascii.ispunct(c)
Checks for any printable ASCII character which is not a space or an alphanumeric character.

curses.ascii.isspace(c)
Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

curses.ascii.isupper(c)
Checks for an ASCII uppercase letter.

curses.ascii.isxdigit(c)
Checks for an ASCII hexadecimal digit. This is equivalent to c in string.hexdigits.

curses.ascii.isctrl(c)
Checks for an ASCII control character (ordinal values 0 to 31).

curses.ascii.ismeta(c)
Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or strings; when the argument is a string, it is first converted using the built-in function ord().

Note that all these functions check ordinal bit values derived from the first character of the string you pass in; they do not actually know anything about the host machine's character encoding. For functions that know about the character encoding (and handle internationalization properly) see the string module.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

curses.ascii.ascii(c)
: Return the ASCII value corresponding to the low 7 bits of c.

curses.ascii.ctrl(c)
: Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

curses.ascii.alt(c)
: Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

curses.ascii.unctrl(c)
: Return a string representation of the ASCII character c. If c is printable, this string is the character itself. If the character is a control character (0x00–0x1f) the string consists of a caret (`'^'`) followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is `'^?'`. If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and `'!'` prepended to the result.

curses.ascii.controlnames
: A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic SP for the space character.

15.14 curses.panel — A panel stack extension for curses

Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.

15.14.1 Functions

The module curses.panel defines the following functions:

curses.panel.bottom_panel()
: Returns the bottom panel in the panel stack.

curses.panel.new_panel(win)
: Returns a panel object, associating it with the given window win. Be aware that you need to keep the returned panel object referenced explicitly. If you don't, the panel object is garbage collected and removed from the panel stack.

curses.panel.top_panel()
: Returns the top panel in the panel stack.

curses.panel.update_panels()
: Updates the virtual screen after changes in the panel stack. This does not call curses.doupdate(), so you'll have to do this yourself.

15.14.2 Panel Objects

Panel objects, as returned by new_panel() above, are windows with a stacking order. There's always a window associated with a panel which determines the content, while the panel methods are responsible for

the window's depth in the panel stack.

Panel objects have the following methods:

Panel.above()
: Returns the panel above the current panel.

Panel.below()
: Returns the panel below the current panel.

Panel.bottom()
: Push the panel to the bottom of the stack.

Panel.hidden()
: Returns true if the panel is hidden (not visible), false otherwise.

Panel.hide()
: Hide the panel. This does not delete the object, it just makes the window on screen invisible.

Panel.move(y, x)
: Move the panel to the screen coordinates (y, x).

Panel.replace(win)
: Change the window associated with the panel to the window win.

Panel.set_userptr(obj)
: Set the panel's user pointer to obj. This is used to associate an arbitrary piece of data with the panel, and can be any Python object.

Panel.show()
: Display the panel (which might have been hidden).

Panel.top()
: Push panel to the top of the stack.

Panel.userptr()
: Returns the user pointer for the panel. This might be any Python object.

Panel.window()
: Returns the window object associated with the panel.

15.15 platform — Access to underlying platform's identifying data

New in version 2.3.

Source code: Lib/platform.py

Note: Specific platforms listed alphabetically, with Linux included in the Unix section.

15.15.1 Cross Platform

platform.architecture(executable=sys.executable, bits='', linkage='')
: Queries the given executable (defaults to the Python interpreter binary) for various architecture information.

Returns a tuple (bits, linkage) which contain information about the bit architecture and the linkage format used for the executable. Both values are returned as strings.

Values that cannot be determined are returned as given by the parameter presets. If bits is given as `''`, the sizeof(pointer) (or sizeof(long) on Python version < 1.5.2) is used as indicator for the supported pointer size.

The function relies on the system's file command to do the actual work. This is available on most if not all Unix platforms and some non-Unix platforms and then only if the executable points to the Python interpreter. Reasonable defaults are used when the above needs are not met.

Note: On Mac OS X (and perhaps other platforms), executable files may be universal files containing multiple architectures.

To get at the "64-bitness" of the current interpreter, it is more reliable to query the sys.maxsize attribute:

```
is_64bits = sys.maxsize > 2**32
```

---

platform.machine()
Returns the machine type, e.g. `'i386'`. An empty string is returned if the value cannot be determined.

platform.node()
Returns the computer's network name (may not be fully qualified!). An empty string is returned if the value cannot be determined.

platform.platform(aliased=0, terse=0)
Returns a single string identifying the underlying platform with as much useful information as possible.

The output is intended to be human readable rather than machine parseable. It may look different on different platforms and this is intended.

If aliased is true, the function will use aliases for various platforms that report system names which differ from their common names, for example SunOS will be reported as Solaris. The system_alias() function is used to implement this.

Setting terse to true causes the function to return only the absolute minimum information needed to identify the platform.

platform.processor()
Returns the (real) processor name, e.g. `'amdk6'`.

An empty string is returned if the value cannot be determined. Note that many platforms do not provide this information or simply return the same value as for machine(). NetBSD does this.

platform.python_build()
Returns a tuple (buildno, builddate) stating the Python build number and date as strings.

platform.python_compiler()
Returns a string identifying the compiler used for compiling Python.

platform.python_branch()
Returns a string identifying the Python implementation SCM branch.

New in version 2.6.

platform.python_implementation()
Returns a string identifying the Python implementation. Possible return values are: 'CPython', 'IronPython', 'Jython', 'PyPy'.

New in version 2.6.

platform.python_revision()
Returns a string identifying the Python implementation SCM revision.

New in version 2.6.

platform.python_version()
Returns the Python version as string 'major.minor.patchlevel'.

Note that unlike the Python sys.version, the returned value will always include the patchlevel (it defaults to 0).

platform.python_version_tuple()
Returns the Python version as tuple (major, minor, patchlevel) of strings.

Note that unlike the Python sys.version, the returned value will always include the patchlevel (it defaults to '0').

platform.release()
Returns the system's release, e.g. '2.2.0' or 'NT' An empty string is returned if the value cannot be determined.

platform.system()
Returns the system/OS name, e.g. 'Linux', 'Windows', or 'Java'. An empty string is returned if the value cannot be determined.

platform.system_alias(system, release, version)
Returns (system, release, version) aliased to common marketing names used for some systems. It also does some reordering of the information in some cases where it would otherwise cause confusion.

platform.version()
Returns the system's release version, e.g. '#3 on degas'. An empty string is returned if the value cannot be determined.

platform.uname()
Fairly portable uname interface. Returns a tuple of strings (system, node, release, version, machine, processor) identifying the underlying platform.

Note that unlike the os.uname() function this also returns possible processor information as additional tuple entry.

Entries which cannot be determined are set to ''.

15.15.2 Java Platform

platform.java_ver(release='', vendor='', vminfo=('', '', ''), osinfo=('', '', ''))
Version interface for Jython.

Returns a tuple (release, vendor, vminfo, osinfo) with vminfo being a tuple (vm_name, vm_release, vm_vendor) and osinfo being a tuple (os_name, os_version, os_arch). Values which cannot be determined are set to the defaults given as parameters (which all default to '').

15.15.3 Windows Platform

platform.win32_ver(release='', version='', csd='', ptype='')
Get additional version information from the Windows Registry and return a tuple (release, version, csd, ptype) referring to OS release, version number, CSD level (service pack) and OS type (multi/single processor).

As a hint: ptype is `'Uniprocessor Free'` on single processor NT machines and `'Multiprocessor Free'` on multi processor machines. The 'Free' refers to the OS version being free of debugging code. It could also state 'Checked' which means the OS version uses debugging code, i.e. code that checks arguments, ranges, etc.

Note: This function works best with Mark Hammond's win32all package installed, but also on Python 2.3 and later (support for this was added in Python 2.6). It obviously only runs on Win32 compatible platforms.

Win95/98 specific

platform.popen(cmd, mode='r', bufsize=None)
Portable popen() interface. Find a working popen implementation preferring win32pipe.popen(). On Windows NT, win32pipe.popen() should work; on Windows 9x it hangs due to bugs in the MS C library.

15.15.4 Mac OS Platform

platform.mac_ver(release='', versioninfo=('', '', ''), machine='')
Get Mac OS version information and return it as tuple (release, versioninfo, machine) with versioninfo being a tuple (version, dev_stage, non_release_version).

Entries which cannot be determined are set to `''`. All tuple entries are strings.

15.15.5 Unix Platforms

platform.dist(distname='', version='', id='', supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...))
This is an old version of the functionality now provided by linux_distribution(). For new code, please use the linux_distribution().

The only difference between the two is that dist() always returns the short name of the distribution taken from the supported_dists parameter.

Deprecated since version 2.6.

platform.linux_distribution(distname='', version='', id='', supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...), full_distribution_name=1)
Tries to determine the name of the Linux OS distribution name.

supported_dists may be given to define the set of Linux distributions to look for. It defaults to a list of currently supported Linux distributions identified by their release file name.

If full_distribution_name is true (default), the full distribution read from the OS is returned. Otherwise the short name taken from supported_dists is used.

Returns a tuple (distname,version,id) which defaults to the args given as parameters. id is the item in parentheses after the version number. It is usually the version codename.

Note: This function is deprecated since Python 3.5 and removed in Python 3.8. See alternative like the distro package.

New in version 2.6.

platform.libc_ver(executable=sys.executable, lib='', version='', chunksize=2048)
Tries to determine the libc version against which the file executable (defaults to the Python interpreter) is linked. Returns a tuple of strings (lib, version) which default to the given parameters in case the lookup fails.

Note that this function has intimate knowledge of how different libc versions add symbols to the executable is probably only usable for executables compiled using gcc.

The file is read and scanned in chunks of chunksize bytes.

15.16 errno — Standard errno system symbols

This module makes available standard errno system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from linux/include/errno.h, which should be pretty all-inclusive.

errno.errorcode
Dictionary providing a mapping from the errno value to the string name in the underlying system. For instance, errno.errorcode[errno.EPERM] maps to `'EPERM'`.

To translate a numeric error code to an error message, use os.strerror().

Of the following list, symbols that are not used on the current platform are not defined by the module. The specific list of defined symbols is available as errno.errorcode.keys(). Symbols available can include:

errno.EPERM
Operation not permitted

errno.ENOENT
No such file or directory

errno.ESRCH
No such process

errno.EINTR
Interrupted system call

errno.EIO
I/O error

errno.ENXIO
No such device or address

errno.E2BIG
Arg list too long

errno.ENOEXEC
Exec format error

errno.EBADF
Bad file number

errno.ECHILD
No child processes

errno.EAGAIN
Try again

errno.ENOMEM
Out of memory

errno.EACCES
: Permission denied

errno.EFAULT
: Bad address

errno.ENOTBLK
: Block device required

errno.EBUSY
: Device or resource busy

errno.EEXIST
: File exists

errno.EXDEV
: Cross-device link

errno.ENODEV
: No such device

errno.ENOTDIR
: Not a directory

errno.EISDIR
: Is a directory

errno.EINVAL
: Invalid argument

errno.ENFILE
: File table overflow

errno.EMFILE
: Too many open files

errno.ENOTTY
: Not a typewriter

errno.ETXTBSY
: Text file busy

errno.EFBIG
: File too large

errno.ENOSPC
: No space left on device

errno.ESPIPE
: Illegal seek

errno.EROFS
: Read-only file system

errno.EMLINK
: Too many links

errno.EPIPE
: Broken pipe

errno.EDOM
: Math argument out of domain of func

errno.ERANGE
Math result not representable

errno.EDEADLK
Resource deadlock would occur

errno.ENAMETOOLONG
File name too long

errno.ENOLCK
No record locks available

errno.ENOSYS
Function not implemented

errno.ENOTEMPTY
Directory not empty

errno.ELOOP
Too many symbolic links encountered

errno.EWOULDBLOCK
Operation would block

errno.ENOMSG
No message of desired type

errno.EIDRM
Identifier removed

errno.ECHRNG
Channel number out of range

errno.EL2NSYNC
Level 2 not synchronized

errno.EL3HLT
Level 3 halted

errno.EL3RST
Level 3 reset

errno.ELNRNG
Link number out of range

errno.EUNATCH
Protocol driver not attached

errno.ENOCSI
No CSI structure available

errno.EL2HLT
Level 2 halted

errno.EBADE
Invalid exchange

errno.EBADR
Invalid request descriptor

errno.EXFULL
Exchange full

errno.ENOANO
No anode

errno.EBADRQC
Invalid request code

errno.EBADSLT
Invalid slot

errno.EDEADLOCK
File locking deadlock error

errno.EBFONT
Bad font file format

errno.ENOSTR
Device not a stream

errno.ENODATA
No data available

errno.ETIME
Timer expired

errno.ENOSR
Out of streams resources

errno.ENONET
Machine is not on the network

errno.ENOPKG
Package not installed

errno.EREMOTE
Object is remote

errno.ENOLINK
Link has been severed

errno.EADV
Advertise error

errno.ESRMNT
Srmount error

errno.ECOMM
Communication error on send

errno.EPROTO
Protocol error

errno.EMULTIHOP
Multihop attempted

errno.EDOTDOT
RFS specific error

errno.EBADMSG
Not a data message

errno.EOVERFLOW
Value too large for defined data type

errno.ENOTUNIQ
Name not unique on network

errno.EBADFD
File descriptor in bad state

errno.EREMCHG
Remote address changed

errno.ELIBACC
Can not access a needed shared library

errno.ELIBBAD
Accessing a corrupted shared library

errno.ELIBSCN
.lib section in a.out corrupted

errno.ELIBMAX
Attempting to link in too many shared libraries

errno.ELIBEXEC
Cannot exec a shared library directly

errno.EILSEQ
Illegal byte sequence

errno.ERESTART
Interrupted system call should be restarted

errno.ESTRPIPE
Streams pipe error

errno.EUSERS
Too many users

errno.ENOTSOCK
Socket operation on non-socket

errno.EDESTADDRREQ
Destination address required

errno.EMSGSIZE
Message too long

errno.EPROTOTYPE
Protocol wrong type for socket

errno.ENOPROTOOPT
Protocol not available

errno.EPROTONOSUPPORT
Protocol not supported

errno.ESOCKTNOSUPPORT
Socket type not supported

errno.EOPNOTSUPP
Operation not supported on transport endpoint

errno.EPFNOSUPPORT
Protocol family not supported

errno.EAFNOSUPPORT
Address family not supported by protocol

errno.EADDRINUSE
Address already in use

errno.EADDRNOTAVAIL
Cannot assign requested address

errno.ENETDOWN
Network is down

errno.ENETUNREACH
Network is unreachable

errno.ENETRESET
Network dropped connection because of reset

errno.ECONNABORTED
Software caused connection abort

errno.ECONNRESET
Connection reset by peer

errno.ENOBUFS
No buffer space available

errno.EISCONN
Transport endpoint is already connected

errno.ENOTCONN
Transport endpoint is not connected

errno.ESHUTDOWN
Cannot send after transport endpoint shutdown

errno.ETOOMANYREFS
Too many references: cannot splice

errno.ETIMEDOUT
Connection timed out

errno.ECONNREFUSED
Connection refused

errno.EHOSTDOWN
Host is down

errno.EHOSTUNREACH
No route to host

errno.EALREADY
Operation already in progress

errno.EINPROGRESS
Operation now in progress

errno.ESTALE
Stale NFS file handle

errno.EUCLEAN
Structure needs cleaning

errno.ENOTNAM
: Not a XENIX named type file

errno.ENAVAIL
: No XENIX semaphores available

errno.EISNAM
: Is a named type file

errno.EREMOTEIO
: Remote I/O error

errno.EDQUOT
: Quota exceeded

15.17 ctypes — A foreign function library for Python

New in version 2.5.

ctypes is a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.

15.17.1 ctypes tutorial

Note: The code samples in this tutorial use doctest to make sure that they actually work. Since some code samples behave differently under Linux, Windows, or Mac OS X, they contain doctest directives in comments.

Note: Some code samples reference the ctypes c_int type. This type is an alias for the c_long type on 32-bit systems. So, you should not be confused if c_long is printed if you would expect c_int — they are actually the same type.

Loading dynamic link libraries

ctypes exports the cdll, and on Windows windll and oledll objects, for loading dynamic link libraries.

You load libraries by accessing them as attributes of these objects. cdll loads libraries which export functions using the standard cdecl calling convention, while windll libraries call functions using the stdcall calling convention. oledll also uses the stdcall calling convention, and assumes the functions return a Windows HRESULT error code. The error code is used to automatically raise a WindowsError exception when the function call fails.

Here are some examples for Windows. Note that msvcrt is the MS standard C library containing most standard C functions, and uses the cdecl calling convention:

```
>>> from ctypes import *
>>> print windll.kernel32
<WinDLL 'kernel32', handle ... at ...>
>>> print cdll.msvcrt
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows appends the usual .dll file suffix automatically.

On Linux, it is required to specify the filename including the extension to load a library, so attribute access can not be used to load libraries. Either the LoadLibrary() method of the dll loaders should be used, or you should load the library by creating an instance of CDLL by calling the constructor:

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

Accessing functions from loaded dlls

Functions are accessed as attributes of dll objects:

```
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print windll.kernel32.GetModuleHandleA
<_FuncPtr object at 0x...>
>>> print windll.kernel32.MyOwnFunction
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

Note that win32 system dlls like kernel32 and user32 often export ANSI as well as UNICODE versions of a function. The UNICODE version is exported with an W appended to the name, while the ANSI version is exported with an A appended to the name. The win32 GetModuleHandle function, which returns a module handle for a given module name, has the following C prototype, and a macro is used to expose one of them as GetModuleHandle depending on whether UNICODE is defined or not:

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll does not try to select one of them by magic, you must access the version you need by specifying GetModuleHandleA or GetModuleHandleW explicitly, and then call it with strings or unicode strings respectively.

Sometimes, dlls export functions with names which aren't valid Python identifiers, like "??2@YAPAXI@Z". In this case you have to use getattr() to retrieve the function:

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

On Windows, some dlls export functions not by name but by ordinal. These functions can be accessed by indexing the dll object with the ordinal number:

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

Calling functions

You can call these functions like any other Python callable. This example uses the time() function, which returns system time in seconds since the Unix epoch, and the GetModuleHandleA() function, which returns a win32 module handle.

This example calls both functions with a NULL pointer (None should be used as the NULL pointer):

```
>>> print libc.time(None)
1150640792
>>> print hex(windll.kernel32.GetModuleHandleA(None))
0x1d000000
>>>
```

ctypes tries to protect you from calling functions with the wrong number of arguments or the wrong calling convention. Unfortunately this only works on Windows. It does this by examining the stack after the function returns, so although an error is raised the function has been called:

```
>>> windll.kernel32.GetModuleHandleA()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>> windll.kernel32.GetModuleHandleA(0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

The same exception is raised when you call an stdcall function with the cdecl calling convention, or vice versa:

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf("spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

To find out the correct calling convention you have to look into the C header file or the documentation for the function you want to call.

On Windows, ctypes uses win32 structured exception handling to prevent crashes from general protection faults when functions are called with invalid argument values:

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
WindowsError: exception: access violation reading 0x00000020
>>>
```

There are, however, enough ways to crash Python with ctypes, so you should be careful anyway.

None, integers, longs, byte strings and unicode strings are the only native Python objects that can directly be used as parameters in these function calls. None is passed as a C NULL pointer, byte strings and unicode strings are passed as pointer to the memory block that contains their data (char * or wchar_t *). Python integers and Python longs are passed as the platforms default C int type, their value is masked to fit into the C type.

Before we move on calling functions with other parameter types, we have to learn more about ctypes data types.

Fundamental data types

ctypes defines a number of primitive C compatible data types:

ctypes type	C type	Python type
c_bool	_Bool	bool (1)
c_char	char	1-character string
c_wchar	wchar_t	1-character unicode string
c_byte	char	int/long
c_ubyte	unsigned char	int/long
c_short	short	int/long
c_ushort	unsigned short	int/long
c_int	int	int/long
c_uint	unsigned int	int/long
c_long	long	int/long
c_ulong	unsigned long	int/long
c_longlong	__int64 or long long	int/long
c_ulonglong	unsigned __int64 or unsigned long long	int/long
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	string or None
c_wchar_p	wchar_t * (NUL terminated)	unicode or None
c_void_p	void *	int/long or None

(1) The constructor accepts any object with a truth value.

All these types can be created by calling them with an optional initializer of the correct type and value:

```
>>> c_int()
c_long(0)
>>> c_char_p("Hello, World")
c_char_p('Hello, World')
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

Since these types are mutable, their value can also be changed afterwards:

```
>>> i = c_int(42)
>>> print i
c_long(42)
>>> print i.value
42
>>> i.value = -99
>>> print i.value
-99
>>>
```

Assigning a new value to instances of the pointer types c_char_p, c_wchar_p, and c_void_p changes the memory location they point to, not the contents of the memory block (of course not, because Python strings are immutable):

```
>>> s = "Hello, World"
>>> c_s = c_char_p(s)
>>> print c_s
c_char_p('Hello, World')
>>> c_s.value = "Hi, there"
>>> print c_s
c_char_p('Hi, there')
>>> print s                 # first string is unchanged
Hello, World
>>>
```

You should be careful, however, not to pass them to functions expecting pointers to mutable memory. If you need mutable memory blocks, ctypes has a create_string_buffer() function which creates these in various ways. The current memory block contents can be accessed (or changed) with the raw property; if you want to access it as NUL terminated string, use the value property:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)      # create a 3 byte buffer, initialized to NUL bytes
>>> print sizeof(p), repr(p.raw)
3 '\x00\x00\x00'
>>> p = create_string_buffer("Hello")      # create a buffer containing a NUL terminated string
>>> print sizeof(p), repr(p.raw)
6 'Hello\x00'
>>> print repr(p.value)
'Hello'
>>> p = create_string_buffer("Hello", 10)  # create a 10 byte buffer
>>> print sizeof(p), repr(p.raw)
10 'Hello\x00\x00\x00\x00\x00\x00'
>>> p.value = "Hi"
>>> print sizeof(p), repr(p.raw)
10 'Hi\x00lo\x00\x00\x00\x00\x00\x00'
>>>
```

The create_string_buffer() function replaces the c_buffer() function (which is still available as an alias), as well as the c_string() function from earlier ctypes releases. To create a mutable memory block containing unicode characters of the C type wchar_t use the create_unicode_buffer() function.

Calling functions, continued

Note that printf prints to the real standard output channel, not to sys.stdout, so these examples will only work at the console prompt, not from within IDLE or PythonWin:

```
>>> printf = libc.printf
>>> printf("Hello, %s\n", "World!")
Hello, World!
14
>>> printf("Hello, %S\n", u"World!")
Hello, World!
14
>>> printf("%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf("%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: Don't know how to convert parameter 2
>>>
```

As has been mentioned before, all Python types except integers, strings, and unicode strings have to be wrapped in their corresponding ctypes type, so that they can be converted to the required C data type:

```
>>> printf("An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

Calling functions with your own custom data types

You can also customize ctypes argument conversion to allow instances of your own classes be used as function arguments. ctypes looks for an _as_parameter_ attribute and uses this as the function argument. Of course, it must be one of integer, string, or unicode:

```
>>> class Bottles(object):
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf("%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

If you don't want to store the instance's data in the _as_parameter_ instance variable, you could define a property() which makes the data available.

Specifying the required argument types (function prototypes)

It is possible to specify the required argument types of functions exported from DLLs by setting the argtypes attribute.

argtypes must be a sequence of C data types (the printf function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature):

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf("String '%s', Int %d, Double %f\n", "Hi", 10, 2.2)
```

(continues on next page)

(continued from previous page)

```
String 'Hi', Int 10, Double 2.200000
37
>>>
```

Specifying a format protects against incompatible argument types (just as a prototype for a C function), and tries to convert the arguments to valid types:

```
>>> printf("%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf("%s %d %f\n", "X", 2, 3)
X 2 3.000000
13
>>>
```

If you have defined your own classes which you pass to function calls, you have to implement a from_param() class method for them to be able to use them in the argtypes sequence. The from_param() class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, its _as_parameter_ attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, unicode, a ctypes instance, or an object with an _as_parameter_ attribute.

Return types

By default functions are assumed to return the C int type. Other return types can be specified by setting the restype attribute of the function object.

Here is a more advanced example, it uses the strchr function, which expects a string pointer and a char, and returns a pointer to a string:

```
>>> strchr = libc.strchr
>>> strchr("abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p   # c_char_p is a pointer to a string
>>> strchr("abcdef", ord("d"))
'def'
>>> print strchr("abcdef", ord("x"))
None
>>>
```

If you want to avoid the ord("x") calls above, you can set the argtypes attribute, and the second argument will be converted from a single character Python string into a C char:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr("abcdef", "d")
'def'
>>> strchr("abcdef", "def")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print strchr("abcdef", "x")
None
>>> strchr("abcdef", "d")
```

(continues on next page)

(continued from previous page)

```
'def'
>>>
```

You can also use a callable Python object (a function or a class for example) as the restype attribute, if the foreign function returns an integer. The callable will be called with the integer the C function returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception:

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
WindowsError: [Errno 126] The specified module could not be found.
>>>
```

WinError is a function which will call Windows FormatMessage() api to get the string representation of an error code, and returns an exception. WinError takes an optional error code parameter, if no one is used, it calls GetLastError() to retrieve it.

Please note that a much more powerful error checking mechanism is available through the errcheck attribute; see the reference manual for details.

Passing pointers (or: passing parameters by reference)

Sometimes a C api function expects a pointer to a data type as parameter, probably to write into the corresponding location, or if the data is too large to be passed by value. This is also known as passing parameters by reference.

ctypes exports the byref() function which is used to pass parameters by reference. The same effect can be achieved with the pointer() function, although pointer() does a lot more work since it constructs a real pointer object, so it is faster to use byref() if you don't need the pointer object in Python itself:

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer('\000' * 32)
>>> print i.value, f.value, repr(s.value)
0 0.0 ''
>>> libc.sscanf("1 3.14 Hello", "%d %f %s",
...             byref(i), byref(f), s)
3
>>> print i.value, f.value, repr(s.value)
1 3.1400001049 'Hello'
>>>
```

Structures and unions

Structures and unions must derive from the Structure and Union base classes which are defined in the ctypes module. Each subclass must define a _fields_ attribute. _fields_ must be a list of 2-tuples, containing a field name and a field type.

The field type must be a ctypes type like c_int, or any other derived ctypes type: structure, union, array, pointer.

Here is a simple example of a POINT structure, which contains two integers named x and y, and also shows how to initialize a structure in the constructor:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print point.x, point.y
10 20
>>> point = POINT(y=5)
>>> print point.x, point.y
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many initializers
>>>
```

You can, however, build much more complicated structures. A structure can itself contain other structures by using a structure as a field type.

Here is a RECT structure which contains two POINTs named upperleft and lowerright:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print rc.upperleft.x, rc.upperleft.y
0 5
>>> print rc.lowerright.x, rc.lowerright.y
0 0
>>>
```

Nested structures can also be initialized in the constructor in several ways:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

Field descriptors can be retrieved from the class, they are useful for debugging because they can provide useful information:

```
>>> print POINT.x
<Field type=c_long, ofs=0, size=4>
>>> print POINT.y
<Field type=c_long, ofs=4, size=4>
>>>
```

Warning: ctypes does not support passing unions or structures with bit-fields to functions by value. While this may work on 32-bit x86, it's not guaranteed by the library to work in the general case. Unions and structures with bit-fields should always be passed to functions by pointer.

Structure/union alignment and byte order

By default, Structure and Union fields are aligned in the same way the C compiler does it. It is possible to override this behavior be specifying a _pack_ class attribute in the subclass definition. This must be set to a positive integer and specifies the maximum alignment for the fields. This is what #pragma pack(n) also does in MSVC.

ctypes uses the native byte order for Structures and Unions. To build structures with non-native byte order, you can use one of the BigEndianStructure, LittleEndianStructure, BigEndianUnion, and LittleEndianUnion base classes. These classes cannot contain pointer fields.

Bit fields in structures and unions

It is possible to create structures and unions containing bit fields. Bit fields are only possible for integer fields, the bit width is specified as the third item in the _fields_ tuples:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print Int.first_16
<Field type=c_long, ofs=0:0, bits=16>
>>> print Int.second_16
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

Arrays

Arrays are sequences, containing a fixed number of instances of the same type.

The recommended way to create array types is by multiplying a data type with a positive integer:

```
TenPointsArrayType = POINT * 10
```

Here is an example of a somewhat artificial data type, a structure containing 4 POINTs among other stuff:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print len(MyStruct().point_array)
4
>>>
```

Instances are created in the usual way, by calling the class:

```
arr = TenPointsArrayType()
for pt in arr:
    print pt.x, pt.y
```

The above code print a series of 0 0 lines, because the array contents is initialized to zeros.

Initializers of the correct type can also be specified:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print ii
<c_long_Array_10 object at 0x...>
>>> for i in ii: print i,
...
1 2 3 4 5 6 7 8 9 10
>>>
```

Pointers

Pointer instances are created by calling the pointer() function on a ctypes type:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

Pointer instances have a contents attribute which returns the object to which the pointer points, the i object above:

```
>>> pi.contents
c_long(42)
>>>
```

Note that ctypes does not have OOR (original object return), it constructs a new, equivalent object each time you retrieve an attribute:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

Assigning another c_int instance to the pointer's contents attribute would cause the pointer to point to the memory location where this is stored:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

Pointer instances can also be indexed with integers:

```
>>> pi[0]
99
>>>
```

Assigning to an integer index changes the pointed to value:

```
>>> print i
c_long(99)
>>> pi[0] = 22
>>> print i
c_long(22)
>>>
```

It is also possible to use indexes different from 0, but you must know what you're doing, just as in C: You can access or change arbitrary memory locations. Generally you only use this feature if you receive a pointer from a C function, and you know that the pointer actually points to an array instead of a single item.

Behind the scenes, the pointer() function does more than simply create pointer instances, it has to create pointer types first. This is done with the POINTER() function, which accepts any ctypes type, and returns a new type:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

Calling the pointer type without an argument creates a NULL pointer. NULL pointers have a False boolean value:

```
>>> null_ptr = POINTER(c_int)()
>>> print bool(null_ptr)
False
>>>
```

ctypes checks for NULL when dereferencing pointers (but dereferencing invalid non-NULL pointers would crash Python):

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

Type conversions

Usually, ctypes does strict type checking. This means, if you have POINTER(c_int) in the argtypes list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for POINTER(c_int), ctypes accepts an array of c_int:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print bar.values[i]
...
1
2
3
>>>
```

In addition, if a function argument is explicitly declared to be a pointer type (such as POINTER(c_int)) in argtypes, an object of the pointed type (c_int in this case) can be passed to the function. ctypes will apply the required byref() conversion in this case automatically.

To set a POINTER type field to NULL, you can assign None:

```
>>> bar.values = None
>>>
```

Sometimes you have instances of incompatible types. In C, you can cast one type into another type. ctypes provides a cast() function which can be used in the same way. The Bar structure defined above accepts POINTER(c_int) pointers or c_int arrays for its values field, but not instances of other types:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

For these cases, the cast() function is handy.

The cast() function can be used to cast a ctypes instance into a pointer to a different ctypes data type. cast() takes two parameters, a ctypes object that is or can be converted to a pointer of some kind, and a ctypes pointer type. It returns an instance of the second argument, which references the same memory block as the first argument:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

So, cast() can be used to assign to the values field of Bar the structure:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
```

(continues on next page)

(continued from previous page)

```
>>> print bar.values[0]
0
>>>
```

Incomplete Types

Incomplete Types are structures, unions or arrays whose members are not yet specified. In C, they are specified by forward declarations, which are defined later:

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

The straightforward translation into ctypes code would be this, but it does not work:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

because the new class cell is not available in the class statement itself. In ctypes, we can define the cell class and set the _fields_ attribute later, after the class statement:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

Lets try it. We create two instances of cell, and let them point to each other, and finally follow the pointer chain a few times:

```
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print p.name,
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

Callback functions

ctypes allows creating C callable function pointers from Python callables. These are sometimes called callback functions.

First, you must create a class for the callback function, the class knows the calling convention, the return type, and the number and types of arguments this function will receive.

The CFUNCTYPE factory function creates types for callback functions using the normal cdecl calling convention, and, on Windows, the WINFUNCTYPE factory function creates types for callback functions using the stdcall calling convention.

Both of these factory functions are called with the result type as first argument, and the callback functions expected argument types as the remaining arguments.

I will present an example here which uses the standard C library's qsort() function, this is used to sort items with the help of a callback function. qsort() will be used to sort an array of integers:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

qsort() must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer else.

So our callback function receives pointers to integers, and must return an integer. First we create the type for the callback function:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

For the first implementation of the callback function, we simply print the arguments we get, and return 0 (incremental development ;-):

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a, b
...     return 0
...
>>>
```

Create the C callable callback:

```
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

And we're ready to go:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
```

(continues on next page)

(continued from previous page)

```
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
>>>
```

We know how to access the contents of a pointer, so lets redefine our callback:

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

Here is what we get on Windows:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 7 1
py_cmp_func 33 1
py_cmp_func 99 1
py_cmp_func 5 1
py_cmp_func 7 5
py_cmp_func 33 5
py_cmp_func 99 5
py_cmp_func 7 99
py_cmp_func 33 99
py_cmp_func 7 33
>>>
```

It is funny to see that on linux the sort function seems to work much more efficiently, it is doing less comparisons:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

Ah, we're nearly done! The last step is to actually compare the two items and return a useful result:

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return a[0] - b[0]
...
>>>
```

Final run on Windows:

```
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 33 7
py_cmp_func 99 33
py_cmp_func 5 99
py_cmp_func 1 99
```

(continues on next page)

(continued from previous page)

```
py_cmp_func 33 7
py_cmp_func 1 33
py_cmp_func 5 33
py_cmp_func 5 7
py_cmp_func 1 7
py_cmp_func 5 1
>>>
```

and on Linux:

```
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

It is quite interesting to see that the Windows qsort() function needs more comparisons than the linux version!

As we can easily check, our array is sorted now:

```
>>> for i in ia: print i,
...
1 5 7 33 99
>>>
```

Note: Make sure you keep references to CFUNCTYPE() objects as long as they are used from C code. ctypes doesn't, and if you don't, they may be garbage collected, crashing your program when a callback is made.

Also, note that if the callback function is called in a thread created outside of Python's control (e.g. by the foreign code that calls the callback), ctypes creates a new dummy Python thread on every invocation. This behavior is correct for most purposes, but it means that values stored with threading.local will not survive across different callbacks, even when those calls are made from the same C thread.

Accessing values exported from dlls

Some shared libraries not only export functions, they also export variables. An example in the Python library itself is the Py_OptimizeFlag, an integer set to 0, 1, or 2, depending on the -O or -OO flag given on startup.

ctypes can access values like this with the in_dll() class methods of the type. pythonapi is a predefined symbol giving access to the Python C api:

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print opt_flag
c_long(0)
>>>
```

If the interpreter would have been started with -O, the sample would have printed c_long(1), or c_long(2) if -OO would have been specified.

An extended example which also demonstrates the use of pointers accesses the PyImport_FrozenModules pointer exported by Python.

Quoting the Python docs: This pointer is initialized to point to an array of "struct _frozen" records, terminated by one whose members are all NULL or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

So manipulating this pointer could even prove useful. To restrict the example size, we show only how this table can be read with ctypes:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int)]
...
>>>
```

We have defined the struct _frozen data type, so we can get the pointer to the table:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

Since table is a pointer to the array of struct_frozen records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the NULL entry:

```
>>> for item in table:
...     print item.name, item.size
...     if item.name is None:
...         break
...
__hello__ 104
__phello__ -104
__phello__.spam 104
None 0
>>>
```

The fact that standard Python has a frozen module and a frozen package (indicated by the negative size member) is not well known, it is only used for testing. Try it out with import __hello__ for example.

Surprises

There are some edge cases in ctypes where you might expect something other than what actually happens.

Consider the following example:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
```

(continues on next page)

(continued from previous page)

```
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
3 4 3 4
>>>
```

Hm. We certainly expected the last statement to print 3 4 1 2. What happened? Here are the steps of the rc.a, rc.b = rc.b, rc.a line above:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

Note that temp0 and temp1 are objects still using the internal buffer of the rc object above. So executing rc.a = temp0 copies the buffer contents of temp0 into rc 's buffer. This, in turn, changes the contents of temp1. So, the last assignment rc.b = temp1, doesn't have the expected effect.

Keep in mind that retrieving sub-objects from Structure, Unions, and Arrays doesn't copy the sub-object, instead it retrieves a wrapper object accessing the root-object's underlying buffer.

Another example that may behave different from what one would expect is this:

```
>>> s = c_char_p()
>>> s.value = "abc def ghi"
>>> s.value
'abc def ghi'
>>> s.value is s.value
False
>>>
```

Why is it printing False? ctypes instances are objects containing a memory block plus some descriptors accessing the contents of the memory. Storing a Python object in the memory block does not store the object itself, instead the contents of the object is stored. Accessing the contents again constructs a new Python object each time!

Variable-sized data types

ctypes provides some support for variable-sized arrays and structures.

The resize() function can be used to resize the memory buffer of an existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a ValueError is raised if this is tried:

```
>>> short_array = (c_short * 4)()
>>> print sizeof(short_array)
8
>>> resize(short_array, 4)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

This is nice and fine, but how would one access the additional elements contained in this array? Since the type still only knows about 4 elements, we get errors accessing other elements:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

Another way to use variable-sized data types with ctypes is to use the dynamic nature of Python, and (re-)define the data type after the required size is already known, on a case by case basis.

15.17.2 ctypes reference

Finding shared libraries

When programming in a compiled language, shared libraries are accessed when compiling/linking a program, and when the program is run.

The purpose of the find_library() function is to locate a library in a way similar to what the compiler does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

The ctypes.util module provides a function which can help to determine the library to load.

ctypes.util.find_library(name)
: Try to find a library and return a pathname. name is the library name without any prefix like lib, suffix like .so, .dylib or version number (this is the form used for the posix linker option -l). If no library can be found, returns None.

The exact functionality is system dependent.

On Linux, find_library() tries to run external programs (/sbin/ldconfig, gcc, and objdump) to find the library file. It returns the filename of the library file. Here are some examples:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

On OS X, find_library() tries several predefined naming schemes and paths to locate the library, and returns a full pathname if successful:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

On Windows, find_library() searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like find_library("c") will fail and return None.

If wrapping a shared library with ctypes, it may be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using find_library() to locate the library at runtime.

Loading shared libraries

There are several ways to load shared libraries into the Python process. One way is to instantiate one of the following classes:

class ctypes.CDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False)
: Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return int.

class ctypes.OleDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False)
: Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the stdcall calling convention, and are assumed to return the windows specific HRESULT code. HRESULT values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an WindowsError is automatically raised.

class ctypes.WinDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False)
: Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the stdcall calling convention, and are assumed to return int by default.

  On Windows CE only the standard calling convention is used, for convenience the WinDLL and OleDLL use the standard calling convention on this platform.

The Python global interpreter lock is released before calling any function exported by these libraries, and reacquired afterwards.

class ctypes.PyDLL(name, mode=DEFAULT_MODE, handle=None)
: Instances of this class behave like CDLL instances, except that the Python GIL is not released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception is raised.

  Thus, this is only useful to call Python C api functions directly.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the handle named parameter, otherwise the underlying platforms dlopen or LoadLibrary function is used to load the library into the process, and to get a handle to it.

The mode parameter can be used to specify how the library is loaded. For details, consult the dlopen(3) manpage. On Windows, mode is ignored. On posix systems, RTLD_NOW is always added, and is not configurable.

The use_errno parameter, when set to true, enables a ctypes mechanism that allows accessing the system errno error number in a safe way. ctypes maintains a thread-local copy of the systems errno variable; if you call foreign functions created with use_errno=True then the errno value before the function call is swapped with the ctypes private copy, the same happens immediately after the function call.

The function ctypes.get_errno() returns the value of the ctypes private copy, and the function ctypes.set_errno() changes the ctypes private copy to a new value and returns the former value.

The use_last_error parameter, when set to true, enables the same mechanism for the Windows error code which is managed by the GetLastError() and SetLastError() Windows API functions; ctypes.get_last_error() and ctypes.set_last_error() are used to request and change the ctypes private copy of the windows error code.

New in version 2.6: The use_last_error and use_errno optional parameters were added.

ctypes.RTLD_GLOBAL
: Flag to use as mode parameter. On platforms where this flag is not available, it is defined as the integer zero.

ctypes.RTLD_LOCAL
: Flag to use as mode parameter. On platforms where this is not available, it is the same as RTLD_GLOBAL.

ctypes.DEFAULT_MODE
: The default mode which is used to load shared libraries. On OSX 10.3, this is RTLD_GLOBAL, otherwise it is the same as RTLD_LOCAL.

Instances of these classes have no public methods. Functions exported by the shared library can be accessed as attributes or by index. Please note that accessing the function through an attribute caches the result and therefore accessing it repeatedly returns the same object each time. On the other hand, accessing it through an index returns a new object each time:

```
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

The following public attributes are available, their name starts with an underscore to not clash with exported function names:

PyDLL._handle
: The system handle used to access the library.

PyDLL._name
: The name of the library passed in the constructor.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the LibraryLoader class, either by calling the LoadLibrary() method, or by retrieving the library as attribute of the loader instance.

class ctypes.LibraryLoader(dlltype)
: Class which loads shared libraries. dlltype should be one of the CDLL, PyDLL, WinDLL, or OleDLL types.

  __getattr__() has special behavior: It allows loading a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

LoadLibrary(name)
: Load a shared library into the process and return it. This method always returns a new instance of the library.

These prefabricated library loaders are available:

ctypes.cdll
: Creates CDLL instances.

ctypes.windll
: Windows only: Creates WinDLL instances.

ctypes.oledll
: Windows only: Creates OleDLL instances.

ctypes.pydll
: Creates PyDLL instances.

For accessing the C Python api directly, a ready-to-use Python shared library object is available:

ctypes.pythonapi
: An instance of PyDLL that exposes Python C API functions as attributes. Note that all these functions are assumed to return C int, which is of course not always the truth, so you have to assign the correct restype attribute to use these functions.

Foreign functions

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any ctypes data instances as arguments, and return the default result type specified by the library loader. They are instances of a private class:

class ctypes._FuncPtr
: Base class for C callable foreign functions.

  Instances of foreign functions are also C compatible data types; they represent C function pointers.

  This behavior can be customized by assigning to special attributes of the foreign function object.

  restype
  : Assign a ctypes type to specify the result type of the foreign function. Use None for void, a function not returning anything.

    It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C int, and the callable will be called with this integer, allowing further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as restype and assign a callable to the errcheck attribute.

  argtypes
  : Assign a tuple of ctypes types to specify the argument types that the function accepts. Functions using the stdcall calling convention can only be called with the same number of arguments as the length of this tuple; functions using the C calling convention accept additional, unspecified arguments as well.

    When a foreign function is called, each actual argument is passed to the from_param() class method of the items in the argtypes tuple, this method allows adapting the actual argument to an object that the foreign function accepts. For example, a c_char_p item in the argtypes tuple will convert a unicode string passed as argument into a byte string using ctypes conversion rules.

New: It is now possible to put items in argtypes which are not ctypes types, but each item must have a from_param() method which returns a value usable as argument (integer, string, ctypes instance). This allows defining adapters that can adapt custom objects as function parameters.

errcheck
Assign a Python function or another callable to this attribute. The callable will be called with three or more arguments:

callable(result, func, arguments)
result is what the foreign function returns, as specified by the restype attribute.

func is the foreign function object itself, this allows reusing the same callable object to check or post process the results of several functions.

arguments is a tuple containing the parameters originally passed to the function call, this allows specializing the behavior on the arguments used.

The object that this function returns will be returned from the foreign function call, but it can also check the result value and raise an exception if the foreign function call failed.

exception ctypes.ArgumentError
This exception is raised when a foreign function call cannot convert one of the passed arguments.

Function prototypes

Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function.

ctypes.CFUNCTYPE(restype, *argtypes, use_errno=False, use_last_error=False)
The returned function prototype creates functions that use the standard C calling convention. The function will release the GIL during the call. If use_errno is set to true, the ctypes private copy of the system errno variable is exchanged with the real errno value before and after the call; use_last_error does the same for the Windows error code.

Changed in version 2.6: The optional use_errno and use_last_error parameters were added.

ctypes.WINFUNCTYPE(restype, *argtypes, use_errno=False, use_last_error=False)
Windows only: The returned function prototype creates functions that use the stdcall calling convention, except on Windows CE where WINFUNCTYPE() is the same as CFUNCTYPE(). The function will release the GIL during the call. use_errno and use_last_error have the same meaning as above.

ctypes.PYFUNCTYPE(restype, *argtypes)
The returned function prototype creates functions that use the Python calling convention. The function will not release the GIL during the call.

Function prototypes created by these factory functions can be instantiated in different ways, depending on the type and number of the parameters in the call:

prototype(address)
Returns a foreign function at the specified address which must be an integer.

prototype(callable)
Create a C callable function (a callback function) from a Python callable.

prototype(func_spec[, paramflags])
Returns a foreign function exported by a shared library. func_spec must be a 2-tuple (name_or_ordinal, library). The first item is the name of the exported function as string,

or the ordinal of the exported function as small integer. The second item is the shared library instance.

prototype(vtbl_index, name[, paramflags[, iid]])
Returns a foreign function that will call a COM method. vtbl_index is the index into the virtual function table, a small non-negative integer. name is name of the COM method. iid is an optional pointer to the interface identifier which is used in extended error reporting.

COM methods use a special calling convention: They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the argtypes tuple.

The optional paramflags parameter creates foreign function wrappers with much more functionality than the features described above.

paramflags must be a tuple of the same length as argtypes.

Each item in this tuple contains further information about a parameter, it must be a tuple containing one, two, or three items.

The first item is an integer containing a combination of direction flags for the parameter:

1 Specifies an input parameter to the function.

2 Output parameter. The foreign function fills in a value.

4 Input parameter which defaults to the integer zero.

The optional second item is the parameter name as string. If this is specified, the foreign function can be called with named parameters.

The optional third item is the default value for this parameter.

This example demonstrates how to wrap the Windows MessageBoxA function so that it supports default parameters and named arguments. The C declaration from the windows header file is this:

```
WINUSERAPI int WINAPI
MessageBoxA(
    HWND hWnd,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType);
```

Here is the wrapping with ctypes:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCSTR, LPCSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", None), (1, "flags", 0)
>>> MessageBox = prototype(("MessageBoxA", windll.user32), paramflags)
>>>
```

The MessageBox foreign function can now be called in these ways:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
>>>
```

A second example demonstrates output parameters. The win32 GetWindowRect function retrieves the dimensions of a specified window by copying them into RECT structure that the caller has to supply. Here is the C declaration:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
    HWND hWnd,
    LPRECT lpRect);
```

Here is the wrapping with ctypes:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

Functions with output parameters will automatically return the output parameter value if there is a single one, or a tuple containing the output parameter values when there are more than one, so the GetWindowRect function now returns a RECT instance, when called.

Output parameters can be combined with the errcheck protocol to do further output processing and error checking. The win32 GetWindowRect api function returns a BOOL to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

If the errcheck function returns the argument tuple it receives unchanged, ctypes continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a RECT instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

Utility functions

ctypes.addressof(obj)
: Returns the address of the memory buffer as integer. obj must be an instance of a ctypes type.

ctypes.alignment(obj_or_type)
: Returns the alignment requirements of a ctypes type. obj_or_type must be a ctypes type or instance.

ctypes.byref(obj[, offset])
: Returns a light-weight pointer to obj, which must be an instance of a ctypes type. offset defaults to zero, and must be an integer that will be added to the internal pointer value.

  byref(obj, offset) corresponds to this C code:

```
(((char *)&obj) + offset)
```

The returned object can only be used as a foreign function call parameter. It behaves similar to pointer(obj), but the construction is a lot faster.

New in version 2.6: The offset optional argument was added.

ctypes.cast(obj, type)
This function is similar to the cast operator in C. It returns a new instance of type which points to the same memory block as obj. type must be a pointer type, and obj must be an object that can be interpreted as a pointer.

ctypes.create_string_buffer(init_or_size[, size])
This function creates a mutable character buffer. The returned object is a ctypes array of c_char.

init_or_size must be an integer which specifies the size of the array, or a string which will be used to initialize the array items.

If a string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the string should not be used.

If the first parameter is a unicode string, it is converted into an 8-bit string according to ctypes conversion rules.

ctypes.create_unicode_buffer(init_or_size[, size])
This function creates a mutable unicode character buffer. The returned object is a ctypes array of c_wchar.

init_or_size must be an integer which specifies the size of the array, or a unicode string which will be used to initialize the array items.

If a unicode string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the string should not be used.

If the first parameter is an 8-bit string, it is converted into a unicode string according to ctypes conversion rules.

ctypes.DllCanUnloadNow()
Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllCanUnloadNow function that the _ctypes extension dll exports.

ctypes.DllGetClassObject()
Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllGetClassObject function that the _ctypes extension dll exports.

ctypes.util.find_library(name)
Try to find a library and return a pathname. name is the library name without any prefix like lib, suffix like .so, .dylib or version number (this is the form used for the posix linker option -l). If no library can be found, returns None.

The exact functionality is system dependent.

Changed in version 2.6: Windows only: find_library("m") or find_library("c") return the result of a call to find_msvcrt().

ctypes.util.find_msvcrt()
Windows only: return the filename of the VC runtime library used by Python, and by the extension modules. If the name of the library cannot be determined, None is returned.

If you need to free memory, for example, allocated by an extension module with a call to the free(void *), it is important that you use the function in the same library that allocated the memory.

New in version 2.6.

ctypes.FormatError([code])
Windows only: Returns a textual description of the error code code. If no error code is specified, the last error code is used by calling the Windows api function GetLastError.

ctypes.GetLastError()
Windows only: Returns the last error code set by Windows in the calling thread. This function calls the Windows GetLastError() function directly, it does not return the ctypes-private copy of the error code.

ctypes.get_errno()
Returns the current value of the ctypes-private copy of the system errno variable in the calling thread.

New in version 2.6.

ctypes.get_last_error()
Windows only: returns the current value of the ctypes-private copy of the system LastError variable in the calling thread.

New in version 2.6.

ctypes.memmove(dst, src, count)
Same as the standard C memmove library function: copies count bytes from src to dst. dst and src must be integers or ctypes instances that can be converted to pointers.

ctypes.memset(dst, c, count)
Same as the standard C memset library function: fills the memory block at address dst with count bytes of value c. dst must be an integer specifying an address, or a ctypes instance.

ctypes.POINTER(type)
This factory function creates and returns a new ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. type must be a ctypes type.

ctypes.pointer(obj)
This function creates a new pointer instance, pointing to obj. The returned object is of the type POINTER(type(obj)).

Note: If you just want to pass a pointer to an object to a foreign function call, you should use byref(obj) which is much faster.

ctypes.resize(obj, size)
This function resizes the internal memory buffer of obj, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by sizeof(type(obj)), but it is possible to enlarge the buffer.

ctypes.set_conversion_mode(encoding, errors)
This function sets the rules that ctypes objects use when converting between 8-bit strings and unicode strings. encoding must be a string specifying an encoding, like 'utf-8' or 'mbcs', errors must be a string specifying the error handling on encoding/decoding errors. Examples of possible values are "strict", "replace", or "ignore".

set_conversion_mode() returns a 2-tuple containing the previous conversion rules. On windows, the initial conversion rules are ('mbcs', 'ignore'), on other systems ('ascii', 'strict').

ctypes.set_errno(value)
Set the current value of the ctypes-private copy of the system errno variable in the calling thread to value and return the previous value.

New in version 2.6.

ctypes.set_last_error(value)
Windows only: set the current value of the ctypes-private copy of the system LastError variable in the calling thread to value and return the previous value.

New in version 2.6.

ctypes.sizeof(obj_or_type)
Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C sizeof operator.

ctypes.string_at(address[, size])
This function returns the string starting at memory address address. If size is specified, it is used as size, otherwise the string is assumed to be zero-terminated.

ctypes.WinError(code=None, descr=None)
Windows only: this function is probably the worst-named thing in ctypes. It creates an instance of WindowsError. If code is not specified, GetLastError is called to determine the error code. If descr is not specified, FormatError() is called to get a textual description of the error.

ctypes.wstring_at(address[, size])
This function returns the wide character string starting at memory address address as unicode string. If size is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.

Data types

class ctypes._CData
This non-public class is the common base class of all ctypes data types. Among other things, all ctypes type instances contain a memory block that hold C compatible data; the address of the memory block is returned by the addressof() helper function. Another instance variable is exposed as _objects; this contains other Python objects that need to be kept alive in case the memory block contains pointers.

Common methods of ctypes data types, these are all class methods (to be exact, they are methods of the metaclass):

from_buffer(source[, offset])
This method returns a ctypes instance that shares the buffer of the source object. The source object must support the writeable buffer interface. The optional offset parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a ValueError is raised.

New in version 2.6.

from_buffer_copy(source[, offset])
This method creates a ctypes instance, copying the buffer from the source object buffer which must be readable. The optional offset parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a ValueError is raised.

New in version 2.6.

from_address(address)
This method returns a ctypes type instance using the memory specified by address which must be an integer.

from_param(obj)
: This method adapts obj to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function's argtypes tuple; it must return an object that can be used as a function call parameter.

  All ctypes data types have a default implementation of this classmethod that normally returns obj if that is an instance of the type. Some types accept other objects as well.

in_dll(library, name)
: This method returns a ctypes type instance exported by a shared library. name is the name of the symbol that exports the data, library is the loaded shared library.

Common instance variables of ctypes data types:

_b_base_
: Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The _b_base_ read-only member is the root ctypes object that owns the memory block.

_b_needsfree_
: This read-only variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

_objects
: This member is either None or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging; never modify the contents of this dictionary.

Fundamental data types

class ctypes._SimpleCData
: This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. _SimpleCData is a subclass of _CData, so it inherits their methods and attributes.

  Changed in version 2.6: ctypes data types that are not and do not contain pointers can now be pickled.

  Instances have a single attribute:

  value
  : This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character string, for character pointer types it is a Python string or unicode string.

    When the value attribute is retrieved from a ctypes instance, usually a new object is returned each time. ctypes does not implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a restype of c_char_p, you will always receive a Python string, not a c_char_p instance.

Subclasses of fundamental data types do not inherit this behavior. So, if a foreign functions restype is a subclass of c_void_p, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the value attribute.

These are the fundamental ctypes data types:

class ctypes.c_byte
Represents the C signed char datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_char
Represents the C char datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.c_char_p
Represents the C char * datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, POINTER(c_char) must be used. The constructor accepts an integer address, or a string.

class ctypes.c_double
Represents the C double datatype. The constructor accepts an optional float initializer.

class ctypes.c_longdouble
Represents the C long double datatype. The constructor accepts an optional float initializer. On platforms where sizeof(long double) == sizeof(double) it is an alias to c_double.

New in version 2.6.

class ctypes.c_float
Represents the C float datatype. The constructor accepts an optional float initializer.

class ctypes.c_int
Represents the C signed int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where sizeof(int) == sizeof(long) it is an alias to c_long.

class ctypes.c_int8
Represents the C 8-bit signed int datatype. Usually an alias for c_byte.

class ctypes.c_int16
Represents the C 16-bit signed int datatype. Usually an alias for c_short.

class ctypes.c_int32
Represents the C 32-bit signed int datatype. Usually an alias for c_int.

class ctypes.c_int64
Represents the C 64-bit signed int datatype. Usually an alias for c_longlong.

class ctypes.c_long
Represents the C signed long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_longlong
Represents the C signed long long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_short
Represents the C signed short datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_size_t
Represents the C size_t datatype.

class ctypes.c_ssize_t
Represents the C ssize_t datatype.

New in version 2.7.

class ctypes.c_ubyte
Represents the C unsigned char datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_uint
Represents the C unsigned int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where sizeof(int) == sizeof(long) it is an alias for c_ulong.

class ctypes.c_uint8
Represents the C 8-bit unsigned int datatype. Usually an alias for c_ubyte.

class ctypes.c_uint16
Represents the C 16-bit unsigned int datatype. Usually an alias for c_ushort.

class ctypes.c_uint32
Represents the C 32-bit unsigned int datatype. Usually an alias for c_uint.

class ctypes.c_uint64
Represents the C 64-bit unsigned int datatype. Usually an alias for c_ulonglong.

class ctypes.c_ulong
Represents the C unsigned long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_ulonglong
Represents the C unsigned long long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_ushort
Represents the C unsigned short datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_void_p
Represents the C void * type. The value is represented as integer. The constructor accepts an optional integer initializer.

class ctypes.c_wchar
Represents the C wchar_t datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.c_wchar_p
Represents the C wchar_t * datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

class ctypes.c_bool
Represent the C bool datatype (more accurately, _Bool from C99). Its value can be True or False, and the constructor accepts any object that has a truth value.

New in version 2.6.

class ctypes.HRESULT
Windows only: Represents a HRESULT value, which contains success or error information for a function or method call.

class ctypes.py_object
Represents the C PyObject * datatype. Calling this without an argument creates a NULL PyObject * pointer.

The ctypes.wintypes module provides quite some other Windows specific data types, for example HWND, WPARAM, or DWORD. Some useful structures like MSG or RECT are also defined.

Structured data types

class ctypes.Union(*args, **kw)
Abstract base class for unions in native byte order.

class ctypes.BigEndianStructure(*args, **kw)
Abstract base class for structures in big endian byte order.

class ctypes.LittleEndianStructure(*args, **kw)
Abstract base class for structures in little endian byte order.

Structures with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

class ctypes.Structure(*args, **kw)
Abstract base class for structures in native byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a _fields_ class variable. ctypes will create descriptors which allow reading and writing the fields by direct attribute accesses. These are the

fields
A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the name of the field, the second item specifies the type of the field; it can be any ctypes data type.

For integer type fields like c_int, a third optional item can be given. It must be a small positive integer defining the bit width of the field.

Field names must be unique within one structure or union. This is not checked, only one field can be accessed when names are repeated.

It is possible to define the _fields_ class variable after the class statement that defines the Structure subclass, this allows creating data types that directly or indirectly reference themselves:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

The _fields_ class variable must, however, be defined before the type is first used (an instance is created, sizeof() is called on it, and so on). Later assignments to the _fields_ class variable will raise an AttributeError.

It is possible to defined sub-subclasses of structure types, they inherit the fields of the base class plus the _fields_ defined in the sub-subclass, if any.

pack
An optional small integer that allows overriding the alignment of structure fields in the instance. _pack_ must already be defined when _fields_ is assigned, otherwise it will have no effect.

anonymous
An optional sequence that lists the names of unnamed (anonymous) fields. _anonymous_ must be already defined when _fields_ is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. ctypes will create descriptors in the structure type that allow accessing the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows):

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

The TYPEDESC structure describes a COM data type, the vt field specifies which one of the union fields is valid. Since the u field is defined as anonymous field, it is now possible to access the members directly off the TYPEDESC instance. td.lptdesc and td.u.lptdesc are equivalent, but the former is faster since it does not need to create a temporary union instance:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to defined sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate _fields_ variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they are appear in _fields_. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize _fields_ with the same name, or create new attributes for names not present in _fields_.

Arrays and pointers

class ctypes.Array(*args)
Abstract base class for arrays.

The recommended way to create concrete array types is by multiplying any ctypes data type with a positive integer. Alternatively, you can subclass this type and define _length_ and _type_ class variables. Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is not itself an Array.

length
A positive integer specifying the number of elements in the array. Out-of-range subscripts result in an IndexError. Will be returned by len().

type
Specifies the type of each element in the array.

Array subclass constructors accept positional arguments, used to initialize the elements in order.

class ctypes._Pointer
Private, abstract base class for pointers.

Concrete pointer types are created by calling POINTER() with the type that will be pointed to; this is done automatically by pointer().

If a pointer points to an array, its elements can be read and written using standard subscript and slice accesses. Pointer objects have no size, so len() will raise TypeError. Negative subscripts will read from the memory before the pointer (as in C), and out-of-range subscripts will probably crash with an access violation (if you're lucky).

type
: Specifies the type pointed to.

contents
: Returns the object to which to pointer points. Assigning to this attribute changes the pointer to point to the assigned object.

CHAPTER

SIXTEEN

OPTIONAL OPERATING SYSTEM SERVICES

The modules described in this chapter provide interfaces to operating system features that are available on selected operating systems only. The interfaces are generally modeled after the Unix or C interfaces but they are available on some other systems as well (e.g. Windows or NT). Here's an overview:

16.1 select — Waiting for I/O completion

This module provides access to the select() and poll() functions available in most operating systems, epoll() available on Linux 2.5+ and kqueue() available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

The module defines the following:

exception select.error
: The exception raised when an error occurs. The accompanying value is a pair containing the numeric error code from errno and the corresponding string, as would be printed by the C function perror().

select.epoll([sizehint=-1])
: (Only supported on Linux 2.5.44 and newer.) Returns an edge polling object, which can be used as Edge or Level Triggered interface for I/O events; see section Edge and Level Trigger Polling (epoll) Objects below for the methods supported by epolling objects.

  New in version 2.6.

select.poll()
: (Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section Polling Objects below for the methods supported by polling objects.

select.kqueue()
: (Only supported on BSD.) Returns a kernel queue object; see section Kqueue Objects below for the methods supported by kqueue objects.

  New in version 2.6.

select.kevent(ident, filter=KQ_FILTER_READ, flags=KQ_EV_ADD, fflags=0, data=0, udata=0)
: (Only supported on BSD.) Returns a kernel event object; see section Kevent Objects below for the methods supported by kevent objects.

  New in version 2.6.

select.select(rlist, wlist, xlist[, timeout])
: This is a straightforward interface to the Unix select() system call. The first three arguments are

sequences of 'waitable objects': either integers representing file descriptors or objects with a parameterless method named fileno() returning such an integer:

- rlist: wait until ready for reading
- wlist: wait until ready for writing
- xlist: wait for an "exceptional condition" (see the manual page for what your system considers such a condition)

Empty sequences are allowed, but acceptance of three empty sequences is platform-dependent. (It is known to work on Unix but not on Windows.) The optional timeout argument specifies a time-out as a floating point number in seconds. When the timeout argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the sequences are Python file objects (e.g. sys.stdin, or objects returned by open() or os.popen()), socket objects returned by socket.socket(). You may also define a wrapper class yourself, as long as it has an appropriate fileno() method (that really returns a file descriptor, not just a random integer).

Note: File objects on Windows are not acceptable, but sockets are. On Windows, the underlying select() function is provided by the WinSock library, and does not handle file descriptors that don't originate from WinSock.

---

select.PIPE_BUF

Files reported as ready for writing by select(), poll() or similar interfaces in this module are guaranteed to not block on a write of up to PIPE_BUF bytes. This value is guaranteed by POSIX to be at least 512. Availability: Unix.

New in version 2.7.

16.1.1 Edge and Level Trigger Polling (epoll) Objects

http://linux.die.net/man/4/epoll

eventmask

Constant	Meaning
EPOLLIN	Available for read
EPOLLOUT	Available for write
EPOLLPRI	Urgent data for read
EPOLLERR	Error condition happened on the assoc. fd
EPOLLHUP	Hang up happened on the assoc. fd
EPOLLET	Set Edge Trigger behavior, the default is Level Trigger behavior
EPOLLONESHOT	Set one-shot behavior. After one event is pulled out, the fd is internally disabled
EPOLLRDNORM	Equivalent to EPOLLIN
EPOLLRDBAND	Priority data band can be read.
EPOLLWRNORM	Equivalent to EPOLLOUT
EPOLLWRBAND	Priority data may be written.
EPOLLMSG	Ignored.

epoll.close()
Close the control file descriptor of the epoll object.

epoll.fileno()
Return the file descriptor number of the control fd.

epoll.fromfd(fd)
Create an epoll object from a given file descriptor.

epoll.register(fd[, eventmask])
Register a fd descriptor with the epoll object.

Note: Registering a file descriptor that's already registered raises an IOError – contrary to Polling Objects's register.

epoll.modify(fd, eventmask)
Modify a register file descriptor.

epoll.unregister(fd)
Remove a registered file descriptor from the epoll object.

epoll.poll([timeout=-1[, maxevents=-1]])
Wait for events. timeout in seconds (float)

16.1.2 Polling Objects

The poll() system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. poll() scales better because the system call only requires listing the file descriptors of interest, while select() builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. select() is O(highest file descriptor), while poll() is O(number of file descriptors).

poll.register(fd[, eventmask])
Register a file descriptor with the polling object. Future calls to the poll() method will then check

whether the file descriptor has any pending I/O events. fd can be either an integer, or an object with a fileno() method that returns an integer. File objects implement fileno(), so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants POLLIN, POLLPRI, and POLLOUT, described in the table below. If not specified, the default value used will check for all 3 types of events.

Constant	Meaning
POLLIN	There is data to read
POLLPRI	There is urgent data to read
POLLOUT	Ready for output: writing will not block
POLLERR	Error condition of some sort
POLLHUP	Hung up
POLLNVAL	Invalid request: descriptor not open

Registering a file descriptor that's already registered is not an error, and has the same effect as registering the descriptor exactly once.

poll.modify(fd, eventmask)
Modifies an already registered fd. This has the same effect as register(fd, eventmask). Attempting to modify a file descriptor that was never registered causes an IOError exception with errno ENOENT to be raised.

New in version 2.6.

poll.unregister(fd)
Remove a file descriptor being tracked by a polling object. Just like the register() method, fd can be an integer or an object with a fileno() method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a KeyError exception to be raised.

poll.poll([timeout])
Polls the set of registered file descriptors, and returns a possibly-empty list containing (fd, event) 2-tuples for the descriptors that have events or errors to report. fd is the file descriptor, and event is a bitmask with bits set for the reported events for that descriptor — POLLIN for waiting input, POLLOUT to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If timeout is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If timeout is omitted, negative, or None, the call will block until there is an event for this poll object.

16.1.3 Kqueue Objects

kqueue.close()
Close the control file descriptor of the kqueue object.

kqueue.fileno()
Return the file descriptor number of the control fd.

kqueue.fromfd(fd)
Create a kqueue object from a given file descriptor.

kqueue.control(changelist, max_events[, timeout=None]) → eventlist
Low level interface to kevent

- changelist must be an iterable of kevent object or None

- max_events must be 0 or a positive integer
- timeout in seconds (floats possible)

16.1.4 Kevent Objects

https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent.ident
: Value used to identify the event. The interpretation depends on the filter but it's usually the file descriptor. In the constructor ident can either be an int or an object with a fileno() function. kevent stores the integer internally.

kevent.filter
: Name of the kernel filter.

Constant	Meaning
KQ_FILTER_READ	Takes a descriptor and returns whenever there is data available to read
KQ_FILTER_WRITE	Takes a descriptor and returns whenever there is data available to write
KQ_FILTER_AIO	AIO requests
KQ_FILTER_VNODE	Returns when one or more of the requested events watched in fflag occurs
KQ_FILTER_PROC	Watch for events on a process id
KQ_FILTER_NETDEV	Watch for events on a network device [not available on Mac OS X]
KQ_FILTER_SIGNAL	Returns whenever the watched signal is delivered to the process
KQ_FILTER_TIMER	Establishes an arbitrary timer

kevent.flags
: Filter action.

Constant	Meaning
KQ_EV_ADD	Adds or modifies an event
KQ_EV_DELETE	Removes an event from the queue
KQ_EV_ENABLE	Permitscontrol() to returns the event
KQ_EV_DISABLE	Disablesevent
KQ_EV_ONESHOT	Removes event after first occurrence
KQ_EV_CLEAR	Reset the state after an event is retrieved
KQ_EV_SYSFLAGS	internal event
KQ_EV_FLAG1	internal event
KQ_EV_EOF	Filter specific EOF condition
KQ_EV_ERROR	See return values

kevent.fflags
: Filter specific flags.

KQ_FILTER_READ and KQ_FILTER_WRITE filter flags:

Constant	Meaning
KQ_NOTE_LOWAT	low water mark of a socket buffer

KQ_FILTER_VNODE filter flags:

Constant	Meaning
KQ_NOTE_DELETE	unlink() was called
KQ_NOTE_WRITE	a write occurred
KQ_NOTE_EXTEND	the file was extended
KQ_NOTE_ATTRIB	an attribute was changed
KQ_NOTE_LINK	the link count has changed
KQ_NOTE_RENAME	the file was renamed
KQ_NOTE_REVOKE	access to the file was revoked

KQ_FILTER_PROC filter flags:

Constant	Meaning
KQ_NOTE_EXIT	the process has exited
KQ_NOTE_FORK	the process has called fork()
KQ_NOTE_EXEC	the process has executed a new process
KQ_NOTE_PCTRLMASK	internal filter flag
KQ_NOTE_PDATAMASK	internal filter flag
KQ_NOTE_TRACK	follow a process across fork()
KQ_NOTE_CHILD	returned on the child process for NOTE_TRACK
KQ_NOTE_TRACKERR	unable to attach to a child

KQ_FILTER_NETDEV filter flags (not available on Mac OS X):

Constant	Meaning
KQ_NOTE_LINKUP	link is up
KQ_NOTE_LINKDOWN	link is down
KQ_NOTE_LINKINV	link state is invalid

kevent.data
: Filter specific data.

kevent.udata
: User defined value.

16.2 threading — Higher-level threading interface

Source code: Lib/threading.py

This module constructs higher-level threading interfaces on top of the lower level thread module. See also the mutex and Queue modules.

The dummy_threading module is provided for situations where threading cannot be used because thread is missing.

Note: Starting with Python 2.6, this module provides PEP 8 compliant aliases and properties to replace the camelCase names that were inspired by Java's threading API. This updated API is compatible with that of the multiprocessing module. However, no schedule has been set for the deprecation of the camelCase names and they remain fully supported in both Python 2.x and 3.x.

Note: Starting with Python 2.5, several Thread methods raise RuntimeError instead of AssertionError if called erroneously.

CPython implementation detail: In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use multiprocessing. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

This module defines the following functions and objects:

threading.active_count()
threading.activeCount()
Return the number of Thread objects currently alive. The returned count is equal to the length of the list returned by enumerate().

Changed in version 2.6: Added active_count() spelling.

threading.Condition()
A factory function that returns a new condition variable object. A condition variable allows one or more threads to wait until they are notified by another thread.

See Condition Objects.

threading.current_thread()
threading.currentThread()
Return the current Thread object, corresponding to the caller's thread of control. If the caller's thread of control was not created through the threading module, a dummy thread object with limited functionality is returned.

Changed in version 2.6: Added current_thread() spelling.

threading.enumerate()
Return a list of all Thread objects currently alive. The list includes daemonic threads, dummy thread objects created by current_thread(), and the main thread. It excludes terminated threads and threads that have not yet been started.

threading.Event()
A factory function that returns a new event object. An event manages a flag that can be set to true with the set() method and reset to false with the clear() method. The wait() method blocks until the flag is true.

See Event Objects.

class threading.local
A class that represents thread-local data. Thread-local data are data whose values are thread specific. To manage thread-local data, just create an instance of local (or a subclass) and store attributes on it:

```
mydata = threading.local()
mydata.x = 1
```

The instance's values will be different for separate threads.

For more details and extensive examples, see the documentation string of the _threading_local module.

New in version 2.4.

threading.Lock()
A factory function that returns a new primitive lock object. Once a thread has acquired it, subsequent attempts to acquire it block, until it is released; any thread may release it.

See Lock Objects.

threading.RLock()
A factory function that returns a new reentrant lock object. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

See RLock Objects.

threading.Semaphore([value])
A factory function that returns a new semaphore object. A semaphore manages a counter representing the number of release() calls minus the number of acquire() calls, plus an initial value. The acquire() method blocks if necessary until it can return without making the counter negative. If not given, value defaults to 1.

See Semaphore Objects.

threading.BoundedSemaphore([value])
A factory function that returns a new bounded semaphore object. A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, ValueError is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, value defaults to 1.

class threading.Thread
A class that represents a thread of control. This class can be safely subclassed in a limited fashion.

See Thread Objects.

class threading.Timer
A thread that executes a function after a specified interval has passed.

See Timer Objects.

threading.settrace(func)
Set a trace function for all threads started from the threading module. The func will be passed to sys.settrace() for each thread, before its run() method is called.

New in version 2.3.

threading.setprofile(func)
Set a profile function for all threads started from the threading module. The func will be passed to sys.setprofile() for each thread, before its run() method is called.

New in version 2.3.

threading.stack_size([size])
Return the thread stack size used when creating new threads. The optional size argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If size is not specified, 0 is used. If changing the thread stack size is unsupported, a ThreadError is raised. If the specified stack size is invalid, a ValueError is raised and the stack size is unmodified. 32kB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size $>$ 32kB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4kB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability: Windows, systems with POSIX threads.

New in version 2.5.

exception threading.ThreadError
Raised for various threading-related errors as described below. Note that many interfaces use RuntimeError instead of ThreadError.

Detailed interfaces for the objects are documented below.

The design of this module is loosely based on Java's threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python. Python's Thread class supports a subset of the behavior of Java's Thread class; currently, there are no priorities, no thread groups, and threads cannot be destroyed, stopped, suspended, resumed, or interrupted. The static methods of Java's Thread class, when implemented, are mapped to module-level functions.

All of the methods described below are executed atomically.

16.2.1 Thread Objects

This class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the run() method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, only override the __init__() and run() methods of this class.

Once a thread object is created, its activity must be started by calling the thread's start() method. This invokes the run() method in a separate thread of control.

Once the thread's activity is started, the thread is considered 'alive'. It stops being alive when its run() method terminates – either normally, or by raising an unhandled exception. The is_alive() method tests whether the thread is alive.

Other threads can call a thread's join() method. This blocks the calling thread until the thread whose join() method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the name attribute.

A thread can be flagged as a "daemon thread". The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the daemon property.

Note: Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully, make them non-daemonic and use a suitable signalling mechanism such as an Event.

There is a "main thread" object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that "dummy thread objects" are created. These are thread objects corresponding to "alien threads", which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive and daemonic, and cannot be join()ed. They are never deleted, since it is impossible to detect the termination of alien threads.

class threading.Thread(group=None, target=None, name=None, args=(), kwargs={})
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

start()
: Start the thread's activity.

  It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

  This method will raise a RuntimeError if called more than once on the same thread object.

run()
: Method representing the thread's activity.

  You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

join([timeout])
: Wait until the thread terminates. This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

  When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

  When the timeout argument is not present or None, the operation will block until the thread terminates.

  A thread can be join()ed many times.

  join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

name
: A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

  New in version 2.6.

getName()
setName()
: Pre-2.6 API for name.

ident
: The 'thread identifier' of this thread or None if the thread has not been started. This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

  New in version 2.6.

is_alive()

isAlive()
: Return whether the thread is alive.

  This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

  Changed in version 2.6: Added is_alive() spelling.

daemon
: A boolean value indicating whether this thread is a daemon thread (True) or not (False). This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

  The entire Python program exits when no alive non-daemon threads are left.

  New in version 2.6.

isDaemon()
setDaemon()
: Pre-2.6 API for daemon.

16.2.2 Lock Objects

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the thread extension module.

A primitive lock is in one of two states, "locked" or "unlocked". It is created in the unlocked state. It has two basic methods, acquire() and release(). When the state is unlocked, acquire() changes the state to locked and returns immediately. When the state is locked, acquire() blocks until a call to release() in another thread changes it to unlocked, then the acquire() call resets it to locked and returns. The release() method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a ThreadError will be raised.

When more than one thread is blocked in acquire() waiting for the state to turn to unlocked, only one thread proceeds when a release() call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

Lock.acquire([blocking])
: Acquire a lock, blocking or non-blocking.

  When invoked with the blocking argument set to True (the default), block until the lock is unlocked, then set it to locked and return True.

  When invoked with the blocking argument set to False, do not block. If a call with blocking set to True would block, return False immediately; otherwise, set the lock to locked and return True.

Lock.release()
: Release a lock.

  When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

  When invoked on an unlocked lock, a ThreadError is raised.

  There is no return value.

16.2.3 RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of "owning thread" and "recursion level" in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

To lock the lock, a thread calls its acquire() method; this returns once the thread owns the lock. To unlock the lock, a thread calls its release() method. acquire()/release() call pairs may be nested; only the final release() (the release() of the outermost pair) resets the lock to unlocked and allows another thread blocked in acquire() to proceed.

RLock.acquire([blocking=1])
Acquire a lock, blocking or non-blocking.

When invoked without arguments: if this thread already owns the lock, increment the recursion level by one, and return immediately. Otherwise, if another thread owns the lock, block until the lock is unlocked. Once the lock is unlocked (not owned by any thread), then grab ownership, set the recursion level to one, and return. If more than one thread is blocked waiting until the lock is unlocked, only one at a time will be able to grab ownership of the lock. There is no return value in this case.

When invoked with the blocking argument set to true, do the same thing as when called without arguments, and return true.

When invoked with the blocking argument set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

RLock.release()
Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. A RuntimeError is raised if this method is called when the lock is unlocked.

There is no return value.

16.2.4 Condition Objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. (Passing one in is useful when several condition variables must share the same lock.)

A condition variable has acquire() and release() methods that call the corresponding methods of the associated lock. It also has a wait() method, and notify() and notifyAll() methods. These three must only be called when the calling thread has acquired the lock, otherwise a RuntimeError is raised.

The wait() method releases the lock, and then blocks until it is awakened by a notify() or notifyAll() call for the same condition variable in another thread. Once awakened, it re-acquires the lock and returns. It is also possible to specify a timeout.

The notify() method wakes up one of the threads waiting for the condition variable, if any are waiting. The notifyAll() method wakes up all threads waiting for the condition variable.

Note: the notify() and notifyAll() methods don't release the lock; this means that the thread or threads awakened will not return from their wait() call immediately, but only when the thread that called notify() or notifyAll() finally relinquishes ownership of the lock.

Tip: the typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call wait() repeatedly until they see the desired state, while threads that modify the state call notify() or notifyAll() when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

```
# Consume one item
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()

# Produce one item
cv.acquire()
make_an_item_available()
cv.notify()
cv.release()
```

To choose between notify() and notifyAll(), consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

class threading.Condition([lock])
: If the lock argument is given and not None, it must be a Lock or RLock object, and it is used as the underlying lock. Otherwise, a new RLock object is created and used as the underlying lock.

  acquire(*args)
  : Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

  release()
  : Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

  wait([timeout])
  : Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a RuntimeError is raised.

    This method releases the underlying lock, and then blocks until it is awakened by a notify() or notifyAll() call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

    When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

    When the underlying lock is an RLock, it is not released using its release() method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the RLock class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

  notify(n=1)
  : By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a RuntimeError is raised.

    This method wakes up at most n of the threads waiting for the condition variable; it is a no-op if no threads are waiting.

The current implementation wakes up exactly n threads, if at least n threads are waiting. However, it's not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than n threads.

Note: an awakened thread does not actually return from its wait() call until it can reacquire the lock. Since notify() does not release the lock, its caller should.

notify_all()
notifyAll()
Wake up all threads waiting on this condition. This method acts like notify(), but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a RuntimeError is raised.

Changed in version 2.6: Added notify_all() spelling.

16.2.5 Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used P() and V() instead of acquire() and release()).

A semaphore manages an internal counter which is decremented by each acquire() call and incremented by each release() call. The counter can never go below zero; when acquire() finds that it is zero, it blocks, waiting until some other thread calls release().

class threading.Semaphore([value])
The optional argument gives the initial value for the internal counter; it defaults to 1. If the value given is less than 0, ValueError is raised.

acquire([blocking])
Acquire a semaphore.

When invoked without arguments: if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called release() to make it larger than zero. This is done with proper interlocking so that if multiple acquire() calls are blocked, release() will wake exactly one of them up. The implementation may pick one at random, so the order in which blocked threads are awakened should not be relied on. There is no return value in this case.

When invoked with blocking set to true, do the same thing as when called without arguments, and return true.

When invoked with blocking set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

release()
Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

Semaphore Example

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore:

```
maxconnections = 5
...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore's acquire and release methods when they need to connect to the server:

```
pool_sema.acquire()
conn = connectdb()
... use connection ...
conn.close()
pool_sema.release()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it's acquired will go undetected.

16.2.6 Event Objects

This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the set() method and reset to false with the clear() method. The wait() method blocks until the flag is true.

class threading.Event
: The internal flag is initially false.

  is_set()
  isSet()
  : Return true if and only if the internal flag is true.

    Changed in version 2.6: Added is_set() spelling.

  set()
  : Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call wait() once the flag is true will not block at all.

  clear()
  : Reset the internal flag to false. Subsequently, threads calling wait() will block until set() is called to set the internal flag to true again.

  wait([timeout])
  : Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls set() to set the flag to true, or until the optional timeout occurs.

    When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

    This method returns the internal flag on exit, so it will always return True except if a timeout is given and the operation times out.

    Changed in version 2.7: Previously, the method always returned None.

16.2.7 Timer Objects

This class represents an action that should be run only after a certain amount of time has passed — a timer. Timer is a subclass of Thread and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their start() method. The timer can be stopped (before its action has begun) by calling the cancel() method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

For example:

```
def hello():
    print "hello, world"

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class threading.Timer(interval, function, args=[], kwargs={})
: Create a timer that will run function with arguments args and keyword arguments kwargs, after interval seconds have passed.

  cancel()
  : Stop the timer, and cancel the execution of the timer's action. This will only work if the timer is still in its waiting stage.

16.2.8 Using locks, conditions, and semaphores in the with statement

All of the objects provided by this module that have acquire() and release() methods can be used as context managers for a with statement. The acquire() method will be called when the block is entered, and release() will be called when the block is exited.

Currently, Lock, RLock, Condition, Semaphore, and BoundedSemaphore objects may be used as with statement context managers. For example:

```
import threading

some_rlock = threading.RLock()

with some_rlock:
    print "some_rlock is locked while this executes"
```

16.2.9 Importing in threaded code

While the import machinery is thread-safe, there are two key restrictions on threaded imports due to inherent limitations in the way that thread-safety is provided:

- Firstly, other than in the main module, an import should not have the side effect of spawning a new thread and then waiting for that thread in any way. Failing to abide by this restriction can lead to a deadlock if the spawned thread directly or indirectly attempts to import a module.
- Secondly, all import attempts must be completed before the interpreter starts shutting itself down. This can be most easily achieved by only performing imports from non-daemon threads created through the threading module. Daemon threads and threads created directly with the thread module will require some other form of synchronization to ensure they do not attempt imports after system shutdown has commenced. Failure to abide by this restriction will lead to intermittent exceptions and crashes during

interpreter shutdown (as the late imports attempt to access machinery which is no longer in a valid state).

16.3 thread — Multiple threads of control

Note: The thread module has been renamed to _thread in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3; however, you should consider using the high-level threading module instead.

This module provides low-level primitives for working with multiple threads (also called light-weight processes or tasks) — multiple threads of control sharing their global data space. For synchronization, simple locks (also called mutexes or binary semaphores) are provided. The threading module provides an easier to use and higher-level threading API built on top of this module.

The module is optional. It is supported on Windows, Linux, SGI IRIX, Solaris 2.x, as well as on systems that have a POSIX thread (a.k.a. "pthread") implementation. For systems lacking the thread module, the dummy_thread module is available. It duplicates this module's interface and can be used as a drop-in replacement.

It defines the following constant and functions:

exception thread.error
Raised on thread-specific errors.

thread.LockType
This is the type of lock objects.

thread.start_new_thread(function, args[, kwargs])
Start a new thread and return its identifier. The thread executes the function function with the argument list args (which must be a tuple). The optional kwargs argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

thread.interrupt_main()
Raise a KeyboardInterrupt exception in the main thread. A subthread can use this function to interrupt the main thread.

New in version 2.3.

thread.exit()
Raise the SystemExit exception. When not caught, this will cause the thread to exit silently.

thread.allocate_lock()
Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

thread.get_ident()
Return the 'thread identifier' of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

thread.stack_size([size])
Return the thread stack size used when creating new threads. The optional size argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32kB). If size is not specified, 0 is used. If changing the thread stack size is unsupported, the error exception is raised. If the specified stack size is invalid, a

ValueError is raised and the stack size is unmodified. 32kB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32kB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4kB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability: Windows, systems with POSIX threads.

New in version 2.5.

Lock objects have the following methods:

lock.acquire([waitflag])
Without the optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock — that's their reason for existence). If the integer waitflag argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as before. The return value is True if the lock is acquired successfully, False if not.

lock.release()
Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

lock.locked()
Return the status of the lock: True if it has been acquired by some thread, False if not.

In addition to these methods, lock objects can also be used via the with statement, e.g.:

```
import thread

a_lock = thread.allocate_lock()

with a_lock:
    print "a_lock is locked while this executes"
```

Caveats:

- Threads interact strangely with interrupts: the KeyboardInterrupt exception will be received by an arbitrary thread. (When the signal module is available, interrupts always go to the main thread.)
- Calling sys.exit() or raising the SystemExit exception is equivalent to calling thread.exit().
- It is not possible to interrupt the acquire() method on a lock — the KeyboardInterrupt exception will happen after the lock has been acquired.
- When the main thread exits, it is system defined whether the other threads survive. On SGI IRIX using the native thread implementation, they survive. On most other systems, they are killed without executing try ... finally clauses or executing object destructors.
- When the main thread exits, it does not do any of its usual cleanup (except that try ... finally clauses are honored), and the standard I/O files are not flushed.

16.4 dummy_threading — Drop-in replacement for the threading module

Source code: Lib/dummy_threading.py

This module provides a duplicate interface to the threading module. It is meant to be imported when the thread module is not provided on a platform.

Suggested usage is:

```
try:
    import threading as _threading
except ImportError:
    import dummy_threading as _threading
```

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.

16.5 dummy_thread — Drop-in replacement for the thread module

Note: The dummy_thread module has been renamed to _dummy_thread in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3; however, you should consider using the high-lever dummy_threading module instead.

Source code: Lib/dummy_thread.py

This module provides a duplicate interface to the thread module. It is meant to be imported when the thread module is not provided on a platform.

Suggested usage is:

```
try:
    import thread as _thread
except ImportError:
    import dummy_thread as _thread
```

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.

16.6 multiprocessing — Process-based "threading" interface

New in version 2.6.

16.6.1 Introduction

multiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

The multiprocessing module also introduces APIs which do not have analogs in the threading module. A prime example of this is the Pool object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). The following example demonstrates the common practice of defining such functions in a module so that child processes can successfully import that module. This basic example of data parallelism using Pool,

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    p = Pool(5)
    print(p.map(f, [1, 2, 3]))
```

will print to standard output

```
[1, 4, 9]
```

The Process class

In multiprocessing, processes are spawned by creating a Process object and then calling its start() method. Process follows the API of threading.Thread. A trivial example of a multiprocess program is

```
from multiprocessing import Process

def f(name):
    print 'hello', name

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

To show the individual process IDs involved, here is an expanded example:

```
from multiprocessing import Process
import os

def info(title):
    print title
    print 'module name:', __name__
    if hasattr(os, 'getppid'):  # only available on Unix
        print 'parent process:', os.getppid()
    print 'process id:', os.getpid()

def f(name):
    info('function f')
    print 'hello', name

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

For an explanation of why (on Windows) the if __name__ == '__main__' part is necessary, see Programming guidelines.

Exchanging objects between processes

multiprocessing supports two types of communication channel between processes:

Queues

The Queue class is a near clone of Queue.Queue. For example:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print q.get()    # prints "[42, None, 'hello']"
    p.join()
```

Queues are thread and process safe.

Pipes

The Pipe() function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print parent_conn.recv()   # prints "[42, None, 'hello']"
    p.join()
```

The two connection objects returned by Pipe() represent the two ends of the pipe. Each connection object has send() and recv() methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the same end of the pipe at the same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.

Synchronization between processes

multiprocessing contains equivalents of all the synchronization primitives from threading. For instance one can use a lock to ensure that only one process prints to standard output at a time:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    print 'hello world', i
    l.release()
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

Without using the lock output from the different processes is liable to get all mixed up.

Sharing state between processes

As mentioned above, when doing concurrent programming it is usually best to avoid using shared state as far as possible. This is particularly true when using multiple processes.

However, if you really do need to use some shared data then multiprocessing provides a couple of ways of doing so.

Shared memory

Data can be stored in a shared memory map using Value or Array. For example, the following code

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print num.value
    print arr[:]
```

will print

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

The `'d'` and `'i'` arguments used when creating num and arr are typecodes of the kind used by the array module: `'d'` indicates a double precision float and `'i'` indicates a signed integer. These shared objects will be process and thread-safe.

For more flexibility in using shared memory one can use the multiprocessing.sharedctypes module which supports the creation of arbitrary ctypes objects allocated from shared memory.

Server process

A manager object returned by Manager() controls a server process which holds Python objects and allows other processes to manipulate them using proxies.

A manager returned by Manager() will support types list, dict, Namespace, Lock, RLock, Semaphore, BoundedSemaphore, Condition, Event, Queue, Value and Array. For example,

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    manager = Manager()

    d = manager.dict()
    l = manager.list(range(10))

    p = Process(target=f, args=(d, l))
    p.start()
    p.join()

    print d
    print l
```

will print

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Server process managers are more flexible than using shared memory objects because they can be made to support arbitrary object types. Also, a single manager can be shared by processes on different computers over a network. They are, however, slower than using shared memory.

Using a pool of workers

The Pool class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways.

For example:

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    pool = Pool(processes=4)              # start 4 worker processes

    # print "[0, 1, 4,..., 81]"
    print pool.map(f, range(10))

    # print same numbers in arbitrary order
    for i in pool.imap_unordered(f, range(10)):
        print i
```

(continues on next page)

(continued from previous page)

```
    # evaluate "f(20)" asynchronously
    res = pool.apply_async(f, (20,))      # runs in *only* one process
    print res.get(timeout=1)              # prints "400"

    # evaluate "os.getpid()" asynchronously
    res = pool.apply_async(os.getpid, ()) # runs in *only* one process
    print res.get(timeout=1)              # prints the PID of that process

    # launching multiple evaluations asynchronously *may* use more processes
    multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
    print [res.get(timeout=1) for res in multiple_results]

    # make a single worker sleep for 10 secs
    res = pool.apply_async(time.sleep, (10,))
    try:
        print res.get(timeout=1)
    except TimeoutError:
        print "We lacked patience and got a multiprocessing.TimeoutError"
```

Note that the methods of a pool should only ever be used by the process which created it.

Note: Functionality within this package requires that the `__main__` module be importable by the children. This is covered in Programming guidelines however it is worth pointing out here. This means that some examples, such as the Pool examples will not work in the interactive interpreter. For example:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(If you try this it will actually output three full tracebacks interleaved in a semi-random fashion, and then you may have to stop the master process somehow.)

16.6.2 Reference

The multiprocessing package mostly replicates the API of the threading module.

Process and exceptions

class multiprocessing.Process(group=None, target=None, name=None, args=(), kwargs={})

Process objects represent activity that is run in a separate process. The Process class has equivalents of all the methods of threading.Thread.

The constructor should always be called with keyword arguments. group should always be None; it exists solely for compatibility with threading.Thread. target is the callable object to be invoked by the run() method. It defaults to None, meaning nothing is called. name is the process name. By default, a unique name is constructed of the form 'Process-N_1:N_2:...:N_k' where N_1,N_2,...,N_k is a sequence of integers whose length is determined by the generation of the process. args is the argument tuple for the target invocation. kwargs is a dictionary of keyword arguments for the target invocation. By default, no arguments are passed to target.

If a subclass overrides the constructor, it must make sure it invokes the base class constructor (Process.__init__()) before doing anything else to the process.

run()
: Method representing the process's activity.

  You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

start()
: Start the process's activity.

  This must be called at most once per process object. It arranges for the object's run() method to be invoked in a separate process.

join([timeout])
: Block the calling thread until the process whose join() method is called terminates or until the optional timeout occurs.

  If timeout is None then there is no timeout.

  A process can be joined many times.

  A process cannot join itself because this would cause a deadlock. It is an error to attempt to join a process before it has been started.

name
: The process's name.

  The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name. The initial name is set by the constructor.

is_alive()
: Return whether the process is alive.

  Roughly, a process object is alive from the moment the start() method returns until the child process terminates.

daemon
: The process's daemon flag, a Boolean value. This must be set before start() is called.

  The initial value is inherited from the creating process.

  When a process exits, it attempts to terminate all of its daemonic child processes.

  Note that a daemonic process is not allowed to create child processes. Otherwise a daemonic process would leave its children orphaned if it gets terminated when its parent process exits. Additionally, these are not Unix daemons or services, they are normal processes that will be terminated (and not joined) if non-daemonic processes have exited.

In addition to the threading.Thread API, Process objects also support the following attributes and methods:

pid
: Return the process ID. Before the process is spawned, this will be None.

exitcode
: The child's exit code. This will be None if the process has not yet terminated. A negative value -N indicates that the child was terminated by signal N.

authkey
: The process's authentication key (a byte string).

  When multiprocessing is initialized the main process is assigned a random string using os.urandom().

  When a Process object is created, it will inherit the authentication key of its parent process, although this may be changed by setting authkey to another byte string.

  See Authentication keys.

terminate()
: Terminate the process. On Unix this is done using the SIGTERM signal; on Windows TerminateProcess() is used. Note that exit handlers and finally clauses, etc., will not be executed.

  Note that descendant processes of the process will not be terminated – they will simply become orphaned.

  > Warning: If this method is used when the associated process is using a pipe or queue then the pipe or queue is liable to become corrupted and may become unusable by other process. Similarly, if the process has acquired a lock or semaphore etc. then terminating it is liable to cause other processes to deadlock.

Note that the start(), join(), is_alive(), terminate() and exitcode methods should only be called by the process that created the process object.

Example usage of some of the methods of Process:

```
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print p, p.is_alive()
<Process(Process-1, initial)> False
>>> p.start()
>>> print p, p.is_alive()
<Process(Process-1, started)> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print p, p.is_alive()
<Process(Process-1, stopped[SIGTERM])> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.BufferTooShort
: Exception raised by Connection.recv_bytes_into() when the supplied buffer object is too small for the message read.

  If e is an instance of BufferTooShort then e.args[0] will give the message as a byte string.

Pipes and Queues

When using multiple processes, one generally uses message passing for communication between processes and avoids having to use any synchronization primitives like locks.

For passing messages one can use Pipe() (for a connection between two processes) or a queue (which allows multiple producers and consumers).

The Queue, multiprocessing.queues.SimpleQueue and JoinableQueue types are multi-producer, multi-consumer FIFO queues modelled on the Queue.Queue class in the standard library. They differ in that Queue lacks the task_done() and join() methods introduced into Python 2.5's Queue.Queue class.

If you use JoinableQueue then you must call JoinableQueue.task_done() for each task removed from the queue or else the semaphore used to count the number of unfinished tasks may eventually overflow, raising an exception.

Note that one can also create a shared queue by using a manager object – see Managers.

Note: multiprocessing uses the usual Queue.Empty and Queue.Full exceptions to signal a timeout. They are not available in the multiprocessing namespace so you need to import them from Queue.

Note: When an object is put on a queue, the object is pickled and a background thread later flushes the pickled data to an underlying pipe. This has some consequences which are a little surprising, but should not cause any practical difficulties – if they really bother you then you can instead use a queue created with a manager.

(1) After putting an object on an empty queue there may be an infinitesimal delay before the queue's empty() method returns False and get_nowait() can return without raising Queue.Empty.

(2) If multiple processes are enqueuing objects, it is possible for the objects to be received at the other end out-of-order. However, objects enqueued by the same process will always be in the expected order with respect to each other.

Warning: If a process is killed using Process.terminate() or os.kill() while it is trying to use a Queue, then the data in the queue is likely to become corrupted. This may cause any other process to get an exception when it tries to use the queue later on.

Warning: As mentioned above, if a child process has put items on a queue (and it has not used JoinableQueue.cancel_join_thread), then that process will not terminate until all buffered items have been flushed to the pipe.

This means that if you try joining that process you may get a deadlock unless you are sure that all items which have been put on the queue have been consumed. Similarly, if the child process is non-daemonic then the parent process may hang on exit when it tries to join all its non-daemonic children.

Note that a queue created using a manager does not have this issue. See Programming guidelines.

For an example of the usage of queues for interprocess communication see Examples.

multiprocessing.Pipe([duplex])
Returns a pair (conn1, conn2) of Connection objects representing the ends of a pipe.

If duplex is True (the default) then the pipe is bidirectional. If duplex is False then the pipe is unidirectional: conn1 can only be used for receiving messages and conn2 can only be used for sending messages.

class multiprocessing.Queue([maxsize])
Returns a process shared queue implemented using a pipe and a few locks/semaphores. When a process first puts an item on the queue a feeder thread is started which transfers objects from a buffer into the pipe.

The usual Queue.Empty and Queue.Full exceptions from the standard library's Queue module are raised to signal timeouts.

Queue implements all the methods of Queue.Queue except for task_done() and join().

qsize()
Return the approximate size of the queue. Because of multithreading/multiprocessing semantics, this number is not reliable.

Note that this may raise NotImplementedError on Unix platforms like Mac OS X where sem_getvalue() is not implemented.

empty()
Return True if the queue is empty, False otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

full()
Return True if the queue is full, False otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

put(obj[, block[, timeout]])
Put obj into the queue. If the optional argument block is True (the default) and timeout is None (the default), block if necessary until a free slot is available. If timeout is a positive number, it blocks at most timeout seconds and raises the Queue.Full exception if no free slot was available within that time. Otherwise (block is False), put an item on the queue if a free slot is immediately available, else raise the Queue.Full exception (timeout is ignored in that case).

put_nowait(obj)
Equivalent to put(obj, False).

get([block[, timeout]])
Remove and return an item from the queue. If optional args block is True (the default) and timeout is None (the default), block if necessary until an item is available. If timeout is a positive number, it blocks at most timeout seconds and raises the Queue.Empty exception if no item was available within that time. Otherwise (block is False), return an item if one is immediately available, else raise the Queue.Empty exception (timeout is ignored in that case).

get_nowait()
Equivalent to get(False).

Queue has a few additional methods not found in Queue.Queue. These methods are usually unnecessary for most code:

close()
Indicate that no more data will be put on this queue by the current process. The background thread will quit once it has flushed all buffered data to the pipe. This is called automatically when the queue is garbage collected.

join_thread()
Join the background thread. This can only be used after close() has been called. It blocks until the background thread exits, ensuring that all data in the buffer has been flushed to the pipe.

By default if a process is not the creator of the queue then on exit it will attempt to join the queue's background thread. The process can call cancel_join_thread() to make join_thread() do nothing.

cancel_join_thread()
Prevent join_thread() from blocking. In particular, this prevents the background thread from being joined automatically when the process exits – see join_thread().

A better name for this method might be allow_exit_without_flush(). It is likely to cause enqueued data to lost, and you almost certainly will not need to use it. It is really only there if you need the current process to exit immediately without waiting to flush enqueued data to the underlying pipe, and you don't care about lost data.

Note: This class's functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the functionality in this class will be disabled, and attempts to instantiate a Queue will result in an ImportError. See bpo-3770 for additional information. The same holds true for any of the specialized queue types listed below.

---

class multiprocessing.queues.SimpleQueue
It is a simplified Queue type, very close to a locked Pipe.

empty()
Return True if the queue is empty, False otherwise.

get()
Remove and return an item from the queue.

put(item)
Put item into the queue.

class multiprocessing.JoinableQueue([maxsize])
JoinableQueue, a Queue subclass, is a queue which additionally has task_done() and join() methods.

task_done()
Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

join()
Block until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, join() unblocks.

Miscellaneous

multiprocessing.active_children()
Return list of all live children of the current process.

Calling this has the side effect of "joining" any processes which have already finished.

multiprocessing.cpu_count()
Return the number of CPUs in the system. May raise NotImplementedError.

multiprocessing.current_process()
Return the Process object corresponding to the current process.

An analogue of threading.current_thread().

multiprocessing.freeze_support()
Add support for when a program which uses multiprocessing has been frozen to produce a Windows executable. (Has been tested with py2exe, PyInstaller and cx_Freeze.)

One needs to call this function straight after the if __name__ == '__main__' line of the main module. For example:

```
from multiprocessing import Process, freeze_support

def f():
    print 'hello world!'

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

If the freeze_support() line is omitted then trying to run the frozen executable will raise RuntimeError.

Calling freeze_support() has no effect when invoked on any operating system other than Windows. In addition, if the module is being run normally by the Python interpreter on Windows (the program has not been frozen), then freeze_support() has no effect.

multiprocessing.set_executable()
Sets the path of the Python interpreter to use when starting a child process. (By default sys.executable is used). Embedders will probably need to do some thing like

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

before they can create child processes. (Windows only)

Note: multiprocessing contains no analogues of threading.active_count(), threading.enumerate(), threading.settrace(), threading.setprofile(), threading.Timer, or threading.local.

Connection Objects

Connection objects allow the sending and receiving of picklable objects or strings. They can be thought of as message oriented connected sockets.

Connection objects are usually created using Pipe – see also Listeners and Clients.

class Connection

send(obj)
Send an object to the other end of the connection which should be read using recv().

The object must be picklable. Very large pickles (approximately 32 MB+, though it depends on the OS) may raise a ValueError exception.

recv()
Return an object sent from the other end of the connection using send(). Blocks until there is something to receive. Raises EOFError if there is nothing left to receive and the other end was closed.

fileno()
: Return the file descriptor or handle used by the connection.

close()
: Close the connection.

  This is called automatically when the connection is garbage collected.

poll([timeout])
: Return whether there is any data available to be read.

  If timeout is not specified then it will return immediately. If timeout is a number then this specifies the maximum time in seconds to block. If timeout is None then an infinite timeout is used.

send_bytes(buffer[, offset[, size]])
: Send byte data from an object supporting the buffer interface as a complete message.

  If offset is given then data is read from that position in buffer. If size is given then that many bytes will be read from buffer. Very large buffers (approximately 32 MB+, though it depends on the OS) may raise a ValueError exception

recv_bytes([maxlength])
: Return a complete message of byte data sent from the other end of the connection as a string. Blocks until there is something to receive. Raises EOFError if there is nothing left to receive and the other end has closed.

  If maxlength is specified and the message is longer than maxlength then IOError is raised and the connection will no longer be readable.

recv_bytes_into(buffer[, offset])
: Read into buffer a complete message of byte data sent from the other end of the connection and return the number of bytes in the message. Blocks until there is something to receive. Raises EOFError if there is nothing left to receive and the other end was closed.

  buffer must be an object satisfying the writable buffer interface. If offset is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of buffer (in bytes).

  If the buffer is too short then a BufferTooShort exception is raised and the complete message is available as e.args[0] where e is the exception instance.

For example:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes('thank you')
>>> a.recv_bytes()
'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

Warning: The Connection.recv() method automatically unpickles the data it receives, which can be a security risk unless you can trust the process which sent the message.

Therefore, unless the connection object was produced using Pipe() you should only use the recv() and send() methods after performing some sort of authentication. See Authentication keys.

Warning: If a process is killed while it is trying to read or write to a pipe then the data in the pipe is likely to become corrupted, because it may become impossible to be sure where the message boundaries lie.

Synchronization primitives

Generally synchronization primitives are not as necessary in a multiprocess program as they are in a multi-threaded program. See the documentation for threading module.

Note that one can also create synchronization primitives by using a manager object – see Managers.

class multiprocessing.BoundedSemaphore([value])
A bounded semaphore object: a close analog of threading.BoundedSemaphore.

A solitary difference from its close analog exists: its acquire method's first argument is named block and it supports an optional second argument timeout, as is consistent with Lock.acquire().

Note: On Mac OS X, this is indistinguishable from Semaphore because sem_getvalue() is not implemented on that platform.

class multiprocessing.Condition([lock])
A condition variable: a clone of threading.Condition.

If lock is specified then it should be a Lock or RLock object from multiprocessing.

class multiprocessing.Event
A clone of threading.Event. This method returns the state of the internal semaphore on exit, so it will always return True except if a timeout is given and the operation times out.

Changed in version 2.7: Previously, the method always returned None.

class multiprocessing.Lock
A non-recursive lock object: a close analog of threading.Lock. Once a process or thread has acquired a lock, subsequent attempts to acquire it from any process or thread will block until it is released; any process or thread may release it. The concepts and behaviors of threading.Lock as it applies to threads are replicated here in multiprocessing.Lock as it applies to either processes or threads, except as noted.

Note that Lock is actually a factory function which returns an instance of multiprocessing.synchronize.Lock initialized with a default context.

Lock supports the context manager protocol and thus may be used in with statements.

acquire(block=True, timeout=None)
Acquire a lock, blocking or non-blocking.

With the block argument set to True (the default), the method call will block until the lock is in an unlocked state, then set it to locked and return True. Note that the name of this first argument differs from that in threading.Lock.acquire().

With the block argument set to False, the method call does not block. If the lock is currently in a locked state, return False; otherwise set the lock to a locked state and return True.

When invoked with a positive, floating-point value for timeout, block for at most the number of seconds specified by timeout as long as the lock can not be acquired. Invocations with a negative value for timeout are equivalent to a timeout of zero. Invocations with a timeout value of None (the default) set the timeout period to infinite. The timeout argument has no practical implications if the block argument is set to False and is thus ignored. Returns True if the lock has been acquired or False if the timeout period has elapsed. Note that the timeout argument does not exist in this method's analog, threading.Lock.acquire().

release()
Release a lock. This can be called from any process or thread, not only the process or thread which originally acquired the lock.

Behavior is the same as in threading.Lock.release() except that when invoked on an unlocked lock, a ValueError is raised.

class multiprocessing.RLock
A recursive lock object: a close analog of threading.RLock. A recursive lock must be released by the process or thread that acquired it. Once a process or thread has acquired a recursive lock, the same process or thread may acquire it again without blocking; that process or thread must release it once for each time it has been acquired.

Note that RLock is actually a factory function which returns an instance of multiprocessing.synchronize.RLock initialized with a default context.

RLock supports the context manager protocol and thus may be used in with statements.

acquire(block=True, timeout=None)
Acquire a lock, blocking or non-blocking.

When invoked with the block argument set to True, block until the lock is in an unlocked state (not owned by any process or thread) unless the lock is already owned by the current process or thread. The current process or thread then takes ownership of the lock (if it does not already have ownership) and the recursion level inside the lock increments by one, resulting in a return value of True. Note that there are several differences in this first argument's behavior compared to the implementation of threading.RLock.acquire(), starting with the name of the argument itself.

When invoked with the block argument set to False, do not block. If the lock has already been acquired (and thus is owned) by another process or thread, the current process or thread does not take ownership and the recursion level within the lock is not changed, resulting in a return value of False. If the lock is in an unlocked state, the current process or thread takes ownership and the recursion level is incremented, resulting in a return value of True.

Use and behaviors of the timeout argument are the same as in Lock.acquire(). Note that the timeout argument does not exist in this method's analog, threading.RLock.acquire().

release()
Release a lock, decrementing the recursion level. If after the decrement the recursion level is zero, reset the lock to unlocked (not owned by any process or thread) and if any other processes or threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling process or thread.

Only call this method when the calling process or thread owns the lock. An AssertionError is raised if this method is called by a process or thread other than the owner or if the lock is in an unlocked (unowned) state. Note that the type of exception raised in this situation differs from the implemented behavior in threading.RLock.release().

class multiprocessing.Semaphore([value])
A semaphore object: a close analog of threading.Semaphore.

A solitary difference from its close analog exists: its acquire method's first argument is named block and it supports an optional second argument timeout, as is consistent with Lock.acquire().

Note: The acquire() method of BoundedSemaphore, Lock, RLock and Semaphore has a timeout parameter not supported by the equivalents in threading. The signature is acquire(block=True, timeout=None) with keyword parameters being acceptable. If block is True and timeout is not None then it specifies a timeout in seconds. If block is False then timeout is ignored.

On Mac OS X, sem_timedwait is unsupported, so calling acquire() with a timeout will emulate that function's behavior using a sleeping loop.

Note: If the SIGINT signal generated by Ctrl-C arrives while the main thread is blocked by a call to BoundedSemaphore.acquire(), Lock.acquire(), RLock.acquire(), Semaphore.acquire(), Condition.acquire() or Condition.wait() then the call will be immediately interrupted and KeyboardInterrupt will be raised.

This differs from the behaviour of threading where SIGINT will be ignored while the equivalent blocking calls are in progress.

Note: Some of this package's functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the multiprocessing.synchronize module will be disabled, and attempts to import it will result in an ImportError. See bpo-3770 for additional information.

Shared ctypes Objects

It is possible to create shared objects using shared memory which can be inherited by child processes.

multiprocessing.Value(typecode_or_type, *args[, lock])
Return a ctypes object allocated from shared memory. By default the return value is actually a synchronized wrapper for the object.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the array module. *args is passed on to the constructor for the type.

If lock is True (the default) then a new recursive lock object is created to synchronize access to the value. If lock is a Lock or RLock object then that will be used to synchronize access to the value. If lock is False then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Operations like += which involve a read and write are not atomic. So if, for instance, you want to atomically increment a shared value it is insufficient to just do

```
counter.value += 1
```

Assuming the associated lock is recursive (which it is by default) you can instead do

```
with counter.get_lock():
    counter.value += 1
```

Note that lock is a keyword-only argument.

multiprocessing.Array(typecode_or_type, size_or_initializer, *, lock=True)
Return a ctypes array allocated from shared memory. By default the return value is actually a synchronized wrapper for the array.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the array module. If size_or_initializer is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, size_or_initializer is a sequence which is used to initialize the array and whose length determines the length of the array.

If lock is True (the default) then a new lock object is created to synchronize access to the value. If lock is a Lock or RLock object then that will be used to synchronize access to the value. If lock is False then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Note that lock is a keyword only argument.

Note that an array of ctypes.c_char has value and raw attributes which allow one to use it to store and retrieve strings.

The multiprocessing.sharedctypes module

The multiprocessing.sharedctypes module provides functions for allocating ctypes objects from shared memory which can be inherited by child processes.

Note: Although it is possible to store a pointer in shared memory remember that this will refer to a location in the address space of a specific process. However, the pointer is quite likely to be invalid in the context of a second process and trying to dereference the pointer from the second process may cause a crash.

---

multiprocessing.sharedctypes.RawArray(typecode_or_type, size_or_initializer)
Return a ctypes array allocated from shared memory.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the array module. If size_or_initializer is an integer then it determines the length of the array, and the array will be initially zeroed. Otherwise size_or_initializer is a sequence which is used to initialize the array and whose length determines the length of the array.

Note that setting and getting an element is potentially non-atomic – use Array() instead to make sure that access is automatically synchronized using a lock.

multiprocessing.sharedctypes.RawValue(typecode_or_type, *args)
Return a ctypes object allocated from shared memory.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the array module. *args is passed on to the constructor for the type.

Note that setting and getting the value is potentially non-atomic – use Value() instead to make sure that access is automatically synchronized using a lock.

Note that an array of ctypes.c_char has value and raw attributes which allow one to use it to store and retrieve strings – see documentation for ctypes.

multiprocessing.sharedctypes.Array(typecode_or_type, size_or_initializer, *args[, lock])
The same as RawArray() except that depending on the value of lock a process-safe synchronization wrapper may be returned instead of a raw ctypes array.

If lock is True (the default) then a new lock object is created to synchronize access to the value. If lock is a Lock or RLock object then that will be used to synchronize access to the value. If lock is False then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Note that lock is a keyword-only argument.

multiprocessing.sharedctypes.Value(typecode_or_type, *args[, lock])
The same as RawValue() except that depending on the value of lock a process-safe synchronization wrapper may be returned instead of a raw ctypes object.

If lock is True (the default) then a new lock object is created to synchronize access to the value. If lock is a Lock or RLock object then that will be used to synchronize access to the value. If lock is False then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Note that lock is a keyword-only argument.

multiprocessing.sharedctypes.copy(obj)
Return a ctypes object allocated from shared memory which is a copy of the ctypes object obj.

multiprocessing.sharedctypes.synchronized(obj[, lock])
Return a process-safe wrapper object for a ctypes object which uses lock to synchronize access. If lock is None (the default) then a multiprocessing.RLock object is created automatically.

A synchronized wrapper will have two methods in addition to those of the object it wraps: get_obj() returns the wrapped object and get_lock() returns the lock object used for synchronization.

Note that accessing the ctypes object through the wrapper can be a lot slower than accessing the raw ctypes object.

The table below compares the syntax for creating shared ctypes objects from shared memory with the normal ctypes syntax. (In the table MyStruct is some subclass of ctypes.Structure.)

ctypes	sharedctypes using type	sharedctypes using typecode
c_double(2.4)	RawValue(c_double, 2.4)	RawValue('d', 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	
(c_short * 7)()	RawArray(c_short, 7)	RawArray('h', 7)
(c_int * 3)(9, 2, 8)	RawArray(c_int, (9, 2, 8))	RawArray('i', (9, 2, 8))

Below is an example where a number of ctypes objects are modified by a child process:

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', 'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print n.value
    print x.value
    print s.value
    print [(a.x, a.y) for a in A]
```

The results printed are

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

Managers

Managers provide a way to create data which can be shared between different processes. A manager object controls a server process which manages shared objects. Other processes can access the shared objects by using proxies.

multiprocessing.Manager()
: Returns a started SyncManager object which can be used for sharing objects between processes. The returned manager object corresponds to a spawned child process and has methods which will create shared objects and return corresponding proxies.

Manager processes will be shutdown as soon as they are garbage collected or their parent process exits. The manager classes are defined in the multiprocessing.managers module:

class multiprocessing.managers.BaseManager([address[, authkey]])
: Create a BaseManager object.

  Once created one should call start() or get_server().serve_forever() to ensure that the manager object refers to a started manager process.

  address is the address on which the manager process listens for new connections. If address is None then an arbitrary one is chosen.

  authkey is the authentication key which will be used to check the validity of incoming connections to the server process. If authkey is None then current_process().authkey. Otherwise authkey is used and it must be a string.

  start([initializer[, initargs]])
  : Start a subprocess to start the manager. If initializer is not None then the subprocess will call initializer(*initargs) when it starts.

  get_server()
  : Returns a Server object which represents the actual server under the control of the Manager. The Server object supports the serve_forever() method:

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey='abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

Server additionally has an address attribute.

connect()
Connect a local manager object to a remote manager process:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 5000), authkey='abc')
>>> m.connect()
```

shutdown()
Stop the process used by the manager. This is only available if start() has been used to start the server process.

This can be called multiple times.

register(typeid[, callable[, proxytype[, exposed[, method_to_typeid[, create_method]]]]])
A classmethod which can be used for registering a type or callable with the manager class.

typeid is a "type identifier" which is used to identify a particular type of shared object. This must be a string.

callable is a callable used for creating objects for this type identifier. If a manager instance will be created using the from_address() classmethod or if the create_method argument is False then this can be left as None.

proxytype is a subclass of BaseProxy which is used to create proxies for shared objects with this typeid. If None then a proxy class is created automatically.

exposed is used to specify a sequence of method names which proxies for this typeid should be allowed to access using BaseProxy._callmethod(). (If exposed is None then proxytype._exposed_ is used instead if it exists.) In the case where no exposed list is specified, all "public methods" of the shared object will be accessible. (Here a "public method" means any attribute which has a __call__() method and whose name does not begin with '_'.)

method_to_typeid is a mapping used to specify the return type of those exposed methods which should return a proxy. It maps method names to typeid strings. (If method_to_typeid is None then proxytype._method_to_typeid_ is used instead if it exists.) If a method's name is not a key of this mapping or if the mapping is None then the object returned by the method will be copied by value.

create_method determines whether a method should be created with name typeid which can be used to tell the server process to create a new shared object and return a proxy for it. By default it is True.

BaseManager instances also have one read-only property:

address
The address used by the manager.

class multiprocessing.managers.SyncManager
A subclass of BaseManager which can be used for the synchronization of processes. Objects of this type are returned by multiprocessing.Manager().

It also supports creation of shared lists and dictionaries.

BoundedSemaphore([value])
: Create a shared threading.BoundedSemaphore object and return a proxy for it.

Condition([lock])
: Create a shared threading.Condition object and return a proxy for it.

  If lock is supplied then it should be a proxy for a threading.Lock or threading.RLock object.

Event()
: Create a shared threading.Event object and return a proxy for it.

Lock()
: Create a shared threading.Lock object and return a proxy for it.

Namespace()
: Create a shared Namespace object and return a proxy for it.

Queue([maxsize])
: Create a shared Queue.Queue object and return a proxy for it.

RLock()
: Create a shared threading.RLock object and return a proxy for it.

Semaphore([value])
: Create a shared threading.Semaphore object and return a proxy for it.

Array(typecode, sequence)
: Create an array and return a proxy for it.

Value(typecode, value)
: Create an object with a writable value attribute and return a proxy for it.

dict()
dict(mapping)
dict(sequence)
: Create a shared dict object and return a proxy for it.

list()
list(sequence)
: Create a shared list object and return a proxy for it.

Note: Modifications to mutable values or items in dict and list proxies will not be propagated through the manager, because the proxy has no way of knowing when its values or items are modified. To modify such an item, you can re-assign the modified object to the container proxy:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# reassigning the dictionary, the proxy is notified of the change
lproxy[0] = d
```

---

class multiprocessing.managers.Namespace
: A type that can register with SyncManager.

A namespace object has no public methods, but does have writable attributes. Its representation shows the values of its attributes.

However, when using a proxy for a namespace object, an attribute beginning with `'_'` will be an attribute of the proxy and not an attribute of the referent:

```
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3    # this is an attribute of the proxy
>>> print Global
Namespace(x=10, y='hello')
```

Customized managers

To create one's own manager, one creates a subclass of BaseManager and uses the register() classmethod to register new types or callables with the manager class. For example:

```
from multiprocessing.managers import BaseManager

class MathsClass(object):
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    manager = MyManager()
    manager.start()
    maths = manager.Maths()
    print maths.add(4, 3)         # prints 7
    print maths.mul(7, 8)         # prints 56
```

Using a remote manager

It is possible to run a manager server on one machine and have clients use it from other machines (assuming that the firewalls involved allow it).

Running the following commands creates a server for a single shared queue which remote clients can access:

```
>>> from multiprocessing.managers import BaseManager
>>> import Queue
>>> queue = Queue.Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey='abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

One client can access the server as follows:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey='abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

Another client can also use it:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey='abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

Local processes can also access that queue, using the code from above on the client to access it remotely:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super(Worker, self).__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey='abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Proxy Objects

A proxy is an object which refers to a shared object which lives (presumably) in a different process. The shared object is said to be the referent of the proxy. Multiple proxy objects may have the same referent.

A proxy object has methods which invoke corresponding methods of its referent (although not every method of the referent will necessarily be available through the proxy). A proxy can usually be used in most of the same ways that its referent can:

```
>>> from multiprocessing import Manager
>>> manager = Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print l
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print repr(l)
<ListProxy object, typeid 'list' at 0x...>
```

(continues on next page)

(continued from previous page)

```
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

Notice that applying str() to a proxy will return the representation of the referent, whereas applying repr() will return the representation of the proxy.

An important feature of proxy objects is that they are picklable so they can be passed between processes. Note, however, that if a proxy is sent to the corresponding manager's process then unpickling it will produce the referent itself. This means, for example, that one shared object can contain a second:

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # referent of a now contains referent of b
>>> print a, b
[[]] []
>>> b.append('hello')
>>> print a, b
[['hello']] ['hello']
```

Note: The proxy types in multiprocessing do nothing to support comparisons by value. So, for instance, we have:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

One should just use a copy of the referent instead when making comparisons.

---

class multiprocessing.managers.BaseProxy

Proxy objects are instances of subclasses of BaseProxy.

_callmethod(methodname[, args[, kwds]])

Call and return the result of a method of the proxy's referent.

If proxy is a proxy whose referent is obj then the expression

```
proxy._callmethod(methodname, args, kwds)
```

will evaluate the expression

```
getattr(obj, methodname)(*args, **kwds)
```

in the manager's process.

The returned value will be a copy of the result of the call or a proxy to a new shared object – see documentation for the method_to_typeid argument of BaseManager.register().

If an exception is raised by the call, then is re-raised by _callmethod(). If some other exception is raised in the manager's process then this is converted into a RemoteError exception and is raised by _callmethod().

Note in particular that an exception will be raised if methodname has not been exposed.

An example of the usage of _callmethod():

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getslice__', (2, 7))   # equiv to `l[2:7]`
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))     # equiv to `l[20]`
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()
: Return a copy of the referent.

  If the referent is unpicklable then this will raise an exception.

__repr__()
: Return a representation of the proxy object.

__str__()
: Return the representation of the referent.

Cleanup

A proxy object uses a weakref callback so that when it gets garbage collected it deregisters itself from the manager which owns its referent.

A shared object gets deleted from the manager process when there are no longer any proxies referring to it.

Process Pools

One can create a pool of processes which will carry out tasks submitted to it with the Pool class.

class multiprocessing.Pool([processes[, initializer[, initargs[, maxtasksperchild]]]])
: A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

  processes is the number of worker processes to use. If processes is None then the number returned by cpu_count() is used. If initializer is not None then each worker process will call initializer(*initargs) when it starts.

  Note that the methods of the pool object should only be called by the process which created the pool.

  New in version 2.7: maxtasksperchild is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default maxtasksperchild is None, which means worker processes will live as long as the pool.

Note: Worker processes within a Pool typically live for the complete duration of the Pool's work queue. A frequent pattern found in other systems (such as Apache, mod_wsgi, etc) to free resources held by workers is to allow a worker within a pool to complete only a set amount of work before being exiting, being cleaned up and a new process spawned to replace the old one. The maxtasksperchild argument to the Pool exposes this ability to the end user.

---

apply(func[, args[, kwds]])
: Equivalent of the apply() built-in function. It blocks until the result is ready, so apply_async()

is better suited for performing work in parallel. Additionally, func is only executed in one of the workers of the pool.

apply_async(func[, args[, kwds[, callback]]])
A variant of the apply() method which returns a result object.

If callback is specified then it should be a callable which accepts a single argument. When the result becomes ready callback is applied to it (unless the call failed). callback should complete immediately since otherwise the thread which handles the results will get blocked.

map(func, iterable[, chunksize])
A parallel equivalent of the map() built-in function (it supports only one iterable argument though). It blocks until the result is ready.

This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting chunksize to a positive integer.

map_async(func, iterable[, chunksize[, callback]])
A variant of the map() method which returns a result object.

If callback is specified then it should be a callable which accepts a single argument. When the result becomes ready callback is applied to it (unless the call failed). callback should complete immediately since otherwise the thread which handles the results will get blocked.

imap(func, iterable[, chunksize])
An equivalent of itertools.imap().

The chunksize argument is the same as the one used by the map() method. For very long iterables using a large value for chunksize can make the job complete much faster than using the default value of 1.

Also if chunksize is 1 then the next() method of the iterator returned by the imap() method has an optional timeout parameter: next(timeout) will raise multiprocessing.TimeoutError if the result cannot be returned within timeout seconds.

imap_unordered(func, iterable[, chunksize])
The same as imap() except that the ordering of the results from the returned iterator should be considered arbitrary. (Only when there is only one worker process is the order guaranteed to be "correct".)

close()
Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.

terminate()
Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected terminate() will be called immediately.

join()
Wait for the worker processes to exit. One must call close() or terminate() before using join().

class multiprocessing.pool.AsyncResult
The class of the result returned by Pool.apply_async() and Pool.map_async().

get([timeout])
Return the result when it arrives. If timeout is not None and the result does not arrive within timeout seconds then multiprocessing.TimeoutError is raised. If the remote call raised an exception then that exception will be reraised by get().

wait([timeout])
: Wait until the result is available or until timeout seconds pass.

ready()
: Return whether the call has completed.

successful()
: Return whether the call completed without raising an exception. Will raise AssertionError if the result is not ready.

The following example demonstrates the use of a pool:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    pool = Pool(processes=4)              # start 4 worker processes

    result = pool.apply_async(f, (10,))   # evaluate "f(10)" asynchronously in a single process
    print result.get(timeout=1)           # prints "100" unless your computer is *very* slow

    print pool.map(f, range(10))          # prints "[0, 1, 4,..., 81]"

    it = pool.imap(f, range(10))
    print it.next()                       # prints "0"
    print it.next()                       # prints "1"
    print it.next(timeout=1)              # prints "4" unless your computer is *very* slow

    result = pool.apply_async(time.sleep, (10,))
    print result.get(timeout=1)           # raises multiprocessing.TimeoutError
```

Listeners and Clients

Usually message passing between processes is done using queues or by using Connection objects returned by Pipe().

However, the multiprocessing.connection module allows some extra flexibility. It basically gives a high level message oriented API for dealing with sockets or Windows named pipes, and also has support for digest authentication using the hmac module.

multiprocessing.connection.deliver_challenge(connection, authkey)
: Send a randomly generated message to the other end of the connection and wait for a reply.

  If the reply matches the digest of the message using authkey as the key then a welcome message is sent to the other end of the connection. Otherwise AuthenticationError is raised.

multiprocessing.connection.answer_challenge(connection, authkey)
: Receive a message, calculate the digest of the message using authkey as the key, and then send the digest back.

  If a welcome message is not received, then AuthenticationError is raised.

multiprocessing.connection.Client(address[, family[, authenticate[, authkey]]])
: Attempt to set up a connection to the listener which is using address address, returning a Connection.

  The type of the connection is determined by family argument, but this can generally be omitted since it can usually be inferred from the format of address. (See Address Formats)

If authenticate is True or authkey is a string then digest authentication is used. The key used for authentication will be either authkey or current_process().authkey) if authkey is None. If authentication fails then AuthenticationError is raised. See Authentication keys.

class multiprocessing.connection.Listener([address[, family[, backlog[, authenticate[, authkey]]]]])
A wrapper for a bound socket or Windows named pipe which is 'listening' for connections.

address is the address to be used by the bound socket or named pipe of the listener object.

Note: If an address of '0.0.0.0' is used, the address will not be a connectable end point on Windows. If you require a connectable end-point, you should use '127.0.0.1'.

---

family is the type of socket (or named pipe) to use. This can be one of the strings `'AF_INET'` (for a TCP socket), `'AF_UNIX'` (for a Unix domain socket) or `'AF_PIPE'` (for a Windows named pipe). Of these only the first is guaranteed to be available. If family is None then the family is inferred from the format of address. If address is also None then a default is chosen. This default is the family which is assumed to be the fastest available. See Address Formats. Note that if family is `'AF_UNIX'` and address is None then the socket will be created in a private temporary directory created using tempfile.mkstemp().

If the listener object uses a socket then backlog (1 by default) is passed to the listen() method of the socket once it has been bound.

If authenticate is True (False by default) or authkey is not None then digest authentication is used.

If authkey is a string then it will be used as the authentication key; otherwise it must be None.

If authkey is None and authenticate is True then current_process().authkey is used as the authentication key. If authkey is None and authenticate is False then no authentication is done. If authentication fails then AuthenticationError is raised. See Authentication keys.

accept()
Accept a connection on the bound socket or named pipe of the listener object and return a Connection object. If authentication is attempted and fails, then AuthenticationError is raised.

close()
Close the bound socket or named pipe of the listener object. This is called automatically when the listener is garbage collected. However it is advisable to call it explicitly.

Listener objects have the following read-only properties:

address
The address which is being used by the Listener object.

last_accepted
The address from which the last accepted connection came. If this is unavailable then it is None.

The module defines the following exceptions:

exception multiprocessing.connection.ProcessError
The base class of all multiprocessing exceptions.

exception multiprocessing.connection.BufferTooShort
Exception raised by Connection.recv_bytes_into() when the supplied buffer object is too small for the message read.

exception multiprocessing.connection.AuthenticationError
Raised when there is an authentication error.

exception multiprocessing.connection.TimeoutError
Raised by methods with a timeout when the timeout expires.

Examples

The following server code creates a listener which uses 'secret password' as an authentication key. It then waits for a connection and sends some data to the client:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)     # family is deduced to be 'AF_INET'
listener = Listener(address, authkey='secret password')

conn = listener.accept()
print 'connection accepted from', listener.last_accepted

conn.send([2.25, None, 'junk', float])

conn.send_bytes('hello')

conn.send_bytes(array('i', [42, 1729]))

conn.close()
listener.close()
```

The following code connects to the server and receives some data from the server:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)
conn = Client(address, authkey='secret password')

print conn.recv()                 # => [2.25, None, 'junk', float]

print conn.recv_bytes()           # => 'hello'

arr = array('i', [0, 0, 0, 0, 0])
print conn.recv_bytes_into(arr)     # => 8
print arr                       # => array('i', [42, 1729, 0, 0, 0])

conn.close()
```

Address Formats

- An 'AF_INET' address is a tuple of the form (hostname, port) where hostname is a string and port is an integer.
- An 'AF_UNIX' address is a string representing a filename on the filesystem.
- An 'AF_PIPE' address is a string of the form r'\\.\pipe{PipeName}'. To use Client() to connect to a named pipe on a remote computer called ServerName one should use an address of the form r'\ServerName\pipe{PipeName}' instead.

Note that any string beginning with two backslashes is assumed by default to be an 'AF_PIPE' address rather than an 'AF_UNIX' address.

Authentication keys

When one uses Connection.recv(), the data received is automatically unpickled. Unfortunately unpickling data from an untrusted source is a security risk. Therefore Listener and Client() use the hmac module to provide digest authentication.

An authentication key is a string which can be thought of as a password: once a connection is established both ends will demand proof that the other knows the authentication key. (Demonstrating that both ends are using the same key does not involve sending the key over the connection.)

If authentication is requested but no authentication key is specified then the return value of current_process().authkey is used (see Process). This value will be automatically inherited by any Process object that the current process creates. This means that (by default) all processes of a multi-process program will share a single authentication key which can be used when setting up connections between themselves.

Suitable authentication keys can also be generated by using os.urandom().

Logging

Some support for logging is available. Note, however, that the logging package does not use process shared locks so it is possible (depending on the handler type) for messages from different processes to get mixed up.

multiprocessing.get_logger()
: Returns the logger used by multiprocessing. If necessary, a new one will be created.

  When first created the logger has level logging.NOTSET and no default handler. Messages sent to this logger will not by default propagate to the root logger.

  Note that on Windows child processes will only inherit the level of the parent process's logger – any other customization of the logger will not be inherited.

multiprocessing.log_to_stderr()
: This function performs a call to get_logger() but in addition to returning the logger created by get_logger, it adds a handler which sends output to sys.stderr using format '[%(levelname)s/%(processName)s] %(message)s'.

Below is an example session with logging turned on:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

In addition to having these two logging functions, the multiprocessing also exposes two additional logging level attributes. These are SUBWARNING and SUBDEBUG. The table below illustrates where theses fit in the normal level hierarchy.

Level	Numeric value
SUBWARNING	25
SUBDEBUG	5

For a full table of logging levels, see the logging module.

These additional logging levels are used primarily for certain debug messages within the multiprocessing module. Below is the same example as above, except with SUBDEBUG enabled:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(multiprocessing.SUBDEBUG)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../pymp-djGBXN/listener-...'
>>> del m
[SUBDEBUG/MainProcess] finalizer calling ...
[INFO/MainProcess] sending shutdown message to manager
[DEBUG/SyncManager-...] manager received shutdown message
[SUBDEBUG/SyncManager-...] calling <Finalize object, callback=unlink, ...
[SUBDEBUG/SyncManager-...] finalizer calling <built-in function unlink> ...
[SUBDEBUG/SyncManager-...] calling <Finalize object, dead>
[SUBDEBUG/SyncManager-...] finalizer calling <function rmtree at 0x5aa730> ...
[INFO/SyncManager-...] manager exiting with exitcode 0
```

The multiprocessing.dummy module

multiprocessing.dummy replicates the API of multiprocessing but is no more than a wrapper around the threading module.

16.6.3 Programming guidelines

There are certain guidelines and idioms which should be adhered to when using multiprocessing.

All platforms

Avoid shared state

> As far as possible one should try to avoid shifting large amounts of data between processes.
>
> It is probably best to stick to using queues or pipes for communication between processes rather than using the lower level synchronization primitives from the threading module.

Picklability

> Ensure that the arguments to the methods of proxies are picklable.

Thread safety of proxies

> Do not use a proxy object from more than one thread unless you protect it with a lock.
>
> (There is never a problem with different processes using the same proxy.)

Joining zombie processes

On Unix when a process finishes but has not been joined it becomes a zombie. There should never be very many because each time a new process starts (or active_children() is called) all completed processes which have not yet been joined will be joined. Also calling a finished process's Process.is_alive will join the process. Even so it is probably good practice to explicitly join all the processes that you start.

Better to inherit than pickle/unpickle

On Windows many types from multiprocessing need to be picklable so that child processes can use them. However, one should generally avoid sending shared objects to other processes using pipes or queues. Instead you should arrange the program so that a process which needs access to a shared resource created elsewhere can inherit it from an ancestor process.

Avoid terminating processes

Using the Process.terminate method to stop a process is liable to cause any shared resources (such as locks, semaphores, pipes and queues) currently being used by the process to become broken or unavailable to other processes.

Therefore it is probably best to only consider using Process.terminate on processes which never use any shared resources.

Joining processes that use queues

Bear in mind that a process that has put items in a queue will wait before terminating until all the buffered items are fed by the "feeder" thread to the underlying pipe. (The child process can call the cancel_join_thread() method of the queue to avoid this behaviour.)

This means that whenever you use a queue you need to make sure that all items which have been put on the queue will eventually be removed before the process is joined. Otherwise you cannot be sure that processes which have put items on the queue will terminate. Remember also that non-daemonic processes will be joined automatically.

An example which will deadlock is the following:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # this deadlocks
    obj = queue.get()
```

A fix here would be to swap the last two lines (or simply remove the p.join() line).

Explicitly pass resources to child processes

On Unix a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

Apart from making the code (potentially) compatible with Windows this also ensures that as long as the child process is still alive the object will not be garbage collected in the parent process. This might be important if some resource is freed when the object is garbage collected in the parent process.

So for instance

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

should be rewritten as

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

Beware of replacing sys.stdin with a "file like object"

multiprocessing originally unconditionally called:

```
os.close(sys.stdin.fileno())
```

in the multiprocessing.Process._bootstrap() method — this resulted in issues with processes-in-processes. This has been changed to:

```
sys.stdin.close()
sys.stdin = open(os.devnull)
```

Which solves the fundamental issue of processes colliding with each other resulting in a bad file descriptor error, but introduces a potential danger to applications which replace sys.stdin() with a "file-like object" with output buffering. This danger is that if multiple processes call close() on this file-like object, it could result in the same data being flushed to the object multiple times, resulting in corruption.

If you write a file-like object and implement your own caching, you can make it fork-safe by storing the pid whenever you append to the cache, and discarding the cache when the pid changes. For example:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

For more information, see bpo-5155, bpo-5313 and bpo-5331

Windows

Since Windows lacks os.fork() it has a few extra restrictions:

More picklability

> Ensure that all arguments to Process.__init__() are picklable. This means, in particular, that bound or unbound methods cannot be used directly as the target argument on Windows — just define a function and use that instead.
>
> Also, if you subclass Process then make sure that instances will be picklable when the Process.start method is called.

Global variables

> Bear in mind that if code run in a child process tries to access a global variable, then the value it sees (if any) may not be the same as the value in the parent process at the time that Process.start was called.
>
> However, global variables which are just module level constants cause no problems.

Safe importing of main module

> Make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects (such a starting a new process).
>
> For example, under Windows running the following module would fail with a RuntimeError:

```
from multiprocessing import Process

def foo():
    print 'hello'

p = Process(target=foo)
p.start()
```

> Instead one should protect the "entry point" of the program by using if __name__ == '__main__': as follows:

```
from multiprocessing import Process, freeze_support

def foo():
    print 'hello'

if __name__ == '__main__':
    freeze_support()
    p = Process(target=foo)
    p.start()
```

> (The freeze_support() line can be omitted if the program will be run normally instead of frozen.)
>
> This allows the newly spawned Python interpreter to safely import the module and then run the module's foo() function.
>
> Similar restrictions apply if a pool or manager is created in the main module.

16.6.4 Examples

Demonstration of how to create and use customized managers and proxies:

```
#
# This module shows how to use arbitrary callables with a subclass of
# `BaseManager`.
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo(object):
    def f(self):
        print 'you called Foo.f()'
    def g(self):
        print 'you called Foo.g()'
    def _h(self):
        print 'you called Foo._h()'

# A simple generator function
def baz():
    for i in xrange(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ('next', '__next__')
    def __iter__(self):
        return self
    def next(self):
        return self._callmethod('next')
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)
```

(continues on next page)

(continued from previous page)

```
##

def test():
    manager = MyManager()
    manager.start()

    print '-' * 20

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print '-' * 20

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print '-' * 20

    it = manager.baz()
    for i in it:
        print '<%d>' % i,
    print

    print '-' * 20

    op = manager.operator()
    print 'op.add(23, 45) =', op.add(23, 45)
    print 'op.pow(2, 94) =', op.pow(2, 94)
    print 'op.getslice(range(10), 2, 6) =', op.getslice(range(10), 2, 6)
    print 'op.repeat(range(5), 3) =', op.repeat(range(5), 3)
    print 'op._exposed_ =', op._exposed_

##

if __name__ == '__main__':
    freeze_support()
    test()
```

Using Pool:

```
#
# A test of `multiprocessing.Pool` class
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import multiprocessing
import time
```

(continues on next page)

(continued from previous page)

```
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

def f(x):
    return 1.0 / (x-5.0)

def pow3(x):
    return x**3

def noop(x):
    pass

#
# Test code
#

def test():
    print 'cpu_count() = %d\n' % multiprocessing.cpu_count()

    #
    # Create pool
    #

    PROCESSES = 4
    print 'Creating pool with %d processes\n' % PROCESSES
    pool = multiprocessing.Pool(PROCESSES)
    print 'pool = %s' % pool
    print

    #
    # Tests
    #

    TASKS = [(mul, (i, 7)) for i in range(10)] + \
```

(continues on next page)

(continued from previous page)

```
        [(plus, (i, 8)) for i in range(10)]

    results = [pool.apply_async(calculate, t) for t in TASKS]
    imap_it = pool.imap(calculatestar, TASKS)
    imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

    print 'Ordered results using pool.apply_async():'
    for r in results:
        print '\t', r.get()
    print

    print 'Ordered results using pool.imap():'
    for x in imap_it:
        print '\t', x
    print

    print 'Unordered results using pool.imap_unordered():'
    for x in imap_unordered_it:
        print '\t', x
    print

    print 'Ordered results using pool.map() --- will block till complete:'
    for x in pool.map(calculatestar, TASKS):
        print '\t', x
    print

    #
    # Simple benchmarks
    #

    N = 100000
    print 'def pow3(x): return x**3'

    t = time.time()
    A = map(pow3, xrange(N))
    print '\tmap(pow3, xrange(%d)):\n\t\t%s seconds' % \
          (N, time.time() - t)

    t = time.time()
    B = pool.map(pow3, xrange(N))
    print '\tpool.map(pow3, xrange(%d)):\n\t\t%s seconds' % \
          (N, time.time() - t)

    t = time.time()
    C = list(pool.imap(pow3, xrange(N), chunksize=N//8))
    print '\tlist(pool.imap(pow3, xrange(%d), chunksize=%d)):\n\t\t%s' \
          ' seconds' % (N, N//8, time.time() - t)

    assert A == B == C, (len(A), len(B), len(C))
    print

    L = [None] * 1000000
    print 'def noop(x): pass'
    print 'L = [None] * 1000000'

    t = time.time()
```

(continues on next page)

(continued from previous page)

```
    A = map(noop, L)
    print '\tmap(noop, L):\n\t\t%s seconds' % \
          (time.time() - t)

    t = time.time()
    B = pool.map(noop, L)
    print '\tpool.map(noop, L):\n\t\t%s seconds' % \
          (time.time() - t)

    t = time.time()
    C = list(pool.imap(noop, L, chunksize=len(L)//8))
    print '\tlist(pool.imap(noop, L, chunksize=%d)):\n\t\t%s seconds' % \
          (len(L)//8, time.time() - t)

    assert A == B == C, (len(A), len(B), len(C))
    print

    del A, B, C, L

    #
    # Test error handling
    #

    print 'Testing error handling:'

    try:
        print pool.apply(f, (5,))
    except ZeroDivisionError:
        print '\tGot ZeroDivisionError as expected from pool.apply()'
    else:
        raise AssertionError('expected ZeroDivisionError')

    try:
        print pool.map(f, range(10))
    except ZeroDivisionError:
        print '\tGot ZeroDivisionError as expected from pool.map()'
    else:
        raise AssertionError('expected ZeroDivisionError')

    try:
        print list(pool.imap(f, range(10)))
    except ZeroDivisionError:
        print '\tGot ZeroDivisionError as expected from list(pool.imap())'
    else:
        raise AssertionError('expected ZeroDivisionError')

    it = pool.imap(f, range(10))
    for i in range(10):
        try:
            x = it.next()
        except ZeroDivisionError:
            if i == 5:
                pass
        except StopIteration:
            break
        else:
```

(continues on next page)

(continued from previous page)

```
        if i == 5:
            raise AssertionError('expected ZeroDivisionError')

    assert i == 9
    print '\tGot ZeroDivisionError as expected from IMapIterator.next()'
    print

    #
    # Testing timeouts
    #

    print 'Testing ApplyResult.get() with timeout:',
    res = pool.apply_async(calculate, TASKS[0])
    while 1:
        sys.stdout.flush()
        try:
            sys.stdout.write('\n\t%s' % res.get(0.02))
            break
        except multiprocessing.TimeoutError:
            sys.stdout.write('.')
    print
    print

    print 'Testing IMapIterator.next() with timeout:',
    it = pool.imap(calculatestar, TASKS)
    while 1:
        sys.stdout.flush()
        try:
            sys.stdout.write('\n\t%s' % it.next(0.02))
        except StopIteration:
            break
        except multiprocessing.TimeoutError:
            sys.stdout.write('.')
    print
    print

    #
    # Testing callback
    #

    print 'Testing callback:'

    A = []
    B = [56, 0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

    r = pool.apply_async(mul, (7, 8), callback=A.append)
    r.wait()

    r = pool.map_async(pow3, range(10), callback=A.extend)
    r.wait()

    if A == B:
        print '\tcallbacks succeeded\n'
    else:
        print '\t*** callbacks failed\n\t\t%s != %s\n' % (A, B)
```

(continues on next page)

(continued from previous page)

```
    #
    # Check there are no outstanding tasks
    #

    assert not pool._cache, 'cache = %r' % pool._cache

    #
    # Check close() methods
    #

    print 'Testing close():'

    for worker in pool._pool:
        assert worker.is_alive()

    result = pool.apply_async(time.sleep, [0.5])
    pool.close()
    pool.join()

    assert result.get() is None

    for worker in pool._pool:
        assert not worker.is_alive()

    print '\tclose() succeeded\n'

    #
    # Check terminate() method
    #

    print 'Testing terminate():'

    pool = multiprocessing.Pool(2)
    DELTA = 0.1
    ignore = pool.apply(pow3, [2])
    results = [pool.apply_async(time.sleep, [DELTA]) for i in range(100)]
    pool.terminate()
    pool.join()

    for worker in pool._pool:
        assert not worker.is_alive()

    print '\tterminate() succeeded\n'

    #
    # Check garbage collection
    #

    print 'Testing garbage collection:'

    pool = multiprocessing.Pool(2)
    DELTA = 0.1
    processes = pool._pool
    ignore = pool.apply(pow3, [2])
    results = [pool.apply_async(time.sleep, [DELTA]) for i in range(100)]
```

(continues on next page)

(continued from previous page)

```
    results = pool = None

    time.sleep(DELTA * 2)

    for worker in processes:
        assert not worker.is_alive()

    print '\tgarbage collection succeeded\n'


if __name__ == '__main__':
    multiprocessing.freeze_support()

    assert len(sys.argv) in (1, 2)

    if len(sys.argv) == 1 or sys.argv[1] == 'processes':
        print ' Using processes '.center(79, '-')
    elif sys.argv[1] == 'threads':
        print ' Using threads '.center(79, '-')
        import multiprocessing.dummy as multiprocessing
    else:
        print 'Usage:\n\t%s [processes | threads]' % sys.argv[0]
        raise SystemExit(2)

    test()
```

Synchronization types like locks, conditions and queues:

```
#
# A test file for the `multiprocessing` package
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import time, sys, random
from Queue import Empty

import multiprocessing                # may get overwritten


#### TEST_VALUE

def value_func(running, mutex):
    random.seed()
    time.sleep(random.random()*4)

    mutex.acquire()
    print '\n\t\t\t' + str(multiprocessing.current_process()) + ' has finished'
    running.value -= 1
    mutex.release()

def test_value():
    TASKS = 10
    running = multiprocessing.Value('i', TASKS)
    mutex = multiprocessing.Lock()
```

(continues on next page)

(continued from previous page)

```
    for i in range(TASKS):
        p = multiprocessing.Process(target=value_func, args=(running, mutex))
        p.start()

    while running.value > 0:
        time.sleep(0.08)
        mutex.acquire()
        print running.value,
        sys.stdout.flush()
        mutex.release()

    print
    print 'No more running processes'


#### TEST_QUEUE

def queue_func(queue):
    for i in range(30):
        time.sleep(0.5 * random.random())
        queue.put(i*i)
    queue.put('STOP')

def test_queue():
    q = multiprocessing.Queue()

    p = multiprocessing.Process(target=queue_func, args=(q,))
    p.start()

    o = None
    while o != 'STOP':
        try:
            o = q.get(timeout=0.3)
            print o,
            sys.stdout.flush()
        except Empty:
            print 'TIMEOUT'

    print


#### TEST_CONDITION

def condition_func(cond):
    cond.acquire()
    print '\t' + str(cond)
    time.sleep(2)
    print '\tchild is notifying'
    print '\t' + str(cond)
    cond.notify()
    cond.release()

def test_condition():
    cond = multiprocessing.Condition()
```

(continues on next page)

(continued from previous page)

```
    p = multiprocessing.Process(target=condition_func, args=(cond,))
    print cond

    cond.acquire()
    print cond
    cond.acquire()
    print cond

    p.start()

    print 'main is waiting'
    cond.wait()
    print 'main has woken up'

    print cond
    cond.release()
    print cond
    cond.release()

    p.join()
    print cond


#### TEST_SEMAPHORE

def semaphore_func(sema, mutex, running):
    sema.acquire()

    mutex.acquire()
    running.value += 1
    print running.value, 'tasks are running'
    mutex.release()

    random.seed()
    time.sleep(random.random()*2)

    mutex.acquire()
    running.value -= 1
    print '%s has finished' % multiprocessing.current_process()
    mutex.release()

    sema.release()

def test_semaphore():
    sema = multiprocessing.Semaphore(3)
    mutex = multiprocessing.RLock()
    running = multiprocessing.Value('i', 0)

    processes = [
        multiprocessing.Process(target=semaphore_func,
                                args=(sema, mutex, running))
        for i in range(10)
        ]

    for p in processes:
        p.start()
```

(continues on next page)

(continued from previous page)

```
    for p in processes:
        p.join()


#### TEST_JOIN_TIMEOUT

def join_timeout_func():
    print '\tchild sleeping'
    time.sleep(5.5)
    print '\n\tchild terminating'

def test_join_timeout():
    p = multiprocessing.Process(target=join_timeout_func)
    p.start()

    print 'waiting for process to finish'

    while 1:
        p.join(timeout=1)
        if not p.is_alive():
            break
        print '.',
        sys.stdout.flush()


#### TEST_EVENT

def event_func(event):
    print '\t%r is waiting' % multiprocessing.current_process()
    event.wait()
    print '\t%r has woken up' % multiprocessing.current_process()

def test_event():
    event = multiprocessing.Event()

    processes = [multiprocessing.Process(target=event_func, args=(event,))
                 for i in range(5)]

    for p in processes:
        p.start()

    print 'main is sleeping'
    time.sleep(2)

    print 'main is setting event'
    event.set()

    for p in processes:
        p.join()


#### TEST_SHAREDVALUES

def sharedvalues_func(values, arrays, shared_values, shared_arrays):
    for i in range(len(values)):
```

(continues on next page)

(continued from previous page)

```
        v = values[i][1]
        sv = shared_values[i].value
        assert v == sv

    for i in range(len(values)):
        a = arrays[i][1]
        sa = list(shared_arrays[i][:])
        assert a == sa

    print 'Tests passed'

def test_sharedvalues():
    values = [
        ('i', 10),
        ('h', -2),
        ('d', 1.25)
        ]
    arrays = [
        ('i', range(100)),
        ('d', [0.25 * i for i in range(100)]),
        ('H', range(1000))
        ]

    shared_values = [multiprocessing.Value(id, v) for id, v in values]
    shared_arrays = [multiprocessing.Array(id, a) for id, a in arrays]

    p = multiprocessing.Process(
        target=sharedvalues_func,
        args=(values, arrays, shared_values, shared_arrays)
        )
    p.start()
    p.join()

    assert p.exitcode == 0


####

def test(namespace=multiprocessing):
    global multiprocessing

    multiprocessing = namespace

    for func in [ test_value, test_queue, test_condition,
                  test_semaphore, test_join_timeout, test_event,
                  test_sharedvalues ]:

        print '\n\t######## %s\n' % func.__name__
        func()

    ignore = multiprocessing.active_children()      # cleanup any old processes
    if hasattr(multiprocessing, '_debug_info'):
        info = multiprocessing._debug_info()
        if info:
            print info
            raise ValueError('there should be no positive refcounts left')
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    multiprocessing.freeze_support()

    assert len(sys.argv) in (1, 2)

    if len(sys.argv) == 1 or sys.argv[1] == 'processes':
        print ' Using processes '.center(79, '-')
        namespace = multiprocessing
    elif sys.argv[1] == 'manager':
        print ' Using processes and a manager '.center(79, '-')
        namespace = multiprocessing.Manager()
        namespace.Process = multiprocessing.Process
        namespace.current_process = multiprocessing.current_process
        namespace.active_children = multiprocessing.active_children
    elif sys.argv[1] == 'threads':
        print ' Using threads '.center(79, '-')
        import multiprocessing.dummy as namespace
    else:
        print 'Usage:\n\t%s [processes | manager | threads]' % sys.argv[0]
        raise SystemExit(2)

    test(namespace)
```

An example showing how to use queues to feed tasks to a collection of worker processes and collect the results:

```
#
# Simple example which uses a pool of workers to carry out some tasks.
#
# Notice that the results will probably not come out of the output
# queue in the same in the same order as the corresponding tasks were
# put on the input queue.  If it is important to get the results back
# in the original order then consider using `Pool.map()` or
# `Pool.imap()` (which will save on the amount of code needed anyway).
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
```

(continues on next page)

(continued from previous page)

```
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print 'Unordered results:'
    for i in range(len(TASKS1)):
        print '\t', done_queue.get()

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print '\t', done_queue.get()

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
```

(continues on next page)

(continued from previous page)

```
        task_queue.put('STOP')


if __name__ == '__main__':
    freeze_support()
    test()
```

An example of how a pool of worker processes can each run a SimpleHTTPServer.HttpServer instance while sharing a single listening socket.

```
#
# Example where a pool of http servers share a single listening socket
#
# On Windows this module depends on the ability to pickle a socket
# object so that the worker processes can inherit a copy of the server
# object.  (We import `multiprocessing.reduction` to enable this pickling.)
#
# Not sure if we should synchronize access to `socket.accept()` method by
# using a process-shared lock -- does not seem to be necessary.
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import os
import sys

from multiprocessing import Process, current_process, freeze_support
from BaseHTTPServer import HTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

if sys.platform == 'win32':
    import multiprocessing.reduction    # make sockets pickable/inheritable


def note(format, *args):
    sys.stderr.write('[%s]\t%s\n' % (current_process().name, format%args))


class RequestHandler(SimpleHTTPRequestHandler):
    # we override log_message() to show which process is handling the request
    def log_message(self, format, *args):
        note(format, *args)

def serve_forever(server):
    note('starting server')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        pass


def runpool(address, number_of_processes):
    # create a single server object -- children will each inherit a copy
    server = HTTPServer(address, RequestHandler)
```

(continues on next page)

(continued from previous page)

```
    # create child processes to act as workers
    for i in range(number_of_processes-1):
        Process(target=serve_forever, args=(server,)).start()

    # main process also acts as a worker
    serve_forever(server)


def test():
    DIR = os.path.join(os.path.dirname(__file__), '..')
    ADDRESS = ('localhost', 8000)
    NUMBER_OF_PROCESSES = 4

    print 'Serving at http://%s:%d using %d worker processes' % \
          (ADDRESS[0], ADDRESS[1], NUMBER_OF_PROCESSES)
    print 'To exit press Ctrl-' + ['C', 'Break'][sys.platform=='win32']

    os.chdir(DIR)
    runpool(ADDRESS, NUMBER_OF_PROCESSES)


if __name__ == '__main__':
    freeze_support()
    test()
```

Some simple benchmarks comparing multiprocessing with threading:

```
#
# Simple benchmarks for the multiprocessing package
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import time, sys, multiprocessing, threading, Queue, gc

if sys.platform == 'win32':
    _timer = time.clock
else:
    _timer = time.time

delta = 1


#### TEST_QUEUESPEED

def queuespeed_func(q, c, iterations):
    a = '0' * 256
    c.acquire()
    c.notify()
    c.release()

    for i in xrange(iterations):
        q.put(a)

    q.put('STOP')
```

(continues on next page)

(continued from previous page)

```
def test_queuespeed(Process, q, c):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        p = Process(target=queuespeed_func, args=(q, c, iterations))
        c.acquire()
        p.start()
        c.wait()
        c.release()

        result = None
        t = _timer()

        while result != 'STOP':
            result = q.get()

        elapsed = _timer() - t

        p.join()

    print iterations, 'objects passed through the queue in', elapsed, 'seconds'
    print 'average number/sec:', iterations/elapsed


#### TEST_PIPESPEED

def pipe_func(c, cond, iterations):
    a = '0' * 256
    cond.acquire()
    cond.notify()
    cond.release()

    for i in xrange(iterations):
        c.send(a)

    c.send('STOP')

def test_pipespeed():
    c, d = multiprocessing.Pipe()
    cond = multiprocessing.Condition()
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        p = multiprocessing.Process(target=pipe_func,
                                    args=(d, cond, iterations))
        cond.acquire()
        p.start()
        cond.wait()
        cond.release()
```

(continues on next page)

(continued from previous page)

```
        result = None
        t = _timer()

        while result != 'STOP':
            result = c.recv()

        elapsed = _timer() - t
        p.join()

    print iterations, 'objects passed through connection in',elapsed,'seconds'
    print 'average number/sec:', iterations/elapsed


#### TEST_SEQSPEED

def test_seqspeed(seq):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        t = _timer()

        for i in xrange(iterations):
            a = seq[5]

        elapsed = _timer()-t

    print iterations, 'iterations in', elapsed, 'seconds'
    print 'average number/sec:', iterations/elapsed


#### TEST_LOCK

def test_lockspeed(l):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        t = _timer()

        for i in xrange(iterations):
            l.acquire()
            l.release()

        elapsed = _timer()-t

    print iterations, 'iterations in', elapsed, 'seconds'
    print 'average number/sec:', iterations/elapsed


#### TEST_CONDITION
```

(continues on next page)

(continued from previous page)

```
def conditionspeed_func(c, N):
    c.acquire()
    c.notify()

    for i in xrange(N):
        c.wait()
        c.notify()

    c.release()

def test_conditionspeed(Process, c):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        c.acquire()
        p = Process(target=conditionspeed_func, args=(c, iterations))
        p.start()

        c.wait()

        t = _timer()

        for i in xrange(iterations):
            c.notify()
            c.wait()

        elapsed = _timer()-t

        c.release()
        p.join()

    print iterations * 2, 'waits in', elapsed, 'seconds'
    print 'average number/sec:', iterations * 2 / elapsed

####

def test():
    manager = multiprocessing.Manager()

    gc.disable()

    print '\n\t######### testing Queue.Queue\n'
    test_queuespeed(threading.Thread, Queue.Queue(),
                    threading.Condition())
    print '\n\t######### testing multiprocessing.Queue\n'
    test_queuespeed(multiprocessing.Process, multiprocessing.Queue(),
                    multiprocessing.Condition())
    print '\n\t######### testing Queue managed by server process\n'
    test_queuespeed(multiprocessing.Process, manager.Queue(),
                    manager.Condition())
    print '\n\t######### testing multiprocessing.Pipe\n'
    test_pipespeed()
```

(continues on next page)

(continued from previous page)

```
    print

    print '\n\t######## testing list\n'
    test_seqspeed(range(10))
    print '\n\t######## testing list managed by server process\n'
    test_seqspeed(manager.list(range(10)))
    print '\n\t######## testing Array("i", ..., lock=False)\n'
    test_seqspeed(multiprocessing.Array('i', range(10), lock=False))
    print '\n\t######## testing Array("i", ..., lock=True)\n'
    test_seqspeed(multiprocessing.Array('i', range(10), lock=True))

    print

    print '\n\t######## testing threading.Lock\n'
    test_lockspeed(threading.Lock())
    print '\n\t######## testing threading.RLock\n'
    test_lockspeed(threading.RLock())
    print '\n\t######## testing multiprocessing.Lock\n'
    test_lockspeed(multiprocessing.Lock())
    print '\n\t######## testing multiprocessing.RLock\n'
    test_lockspeed(multiprocessing.RLock())
    print '\n\t######## testing lock managed by server process\n'
    test_lockspeed(manager.Lock())
    print '\n\t######## testing rlock managed by server process\n'
    test_lockspeed(manager.RLock())

    print

    print '\n\t######## testing threading.Condition\n'
    test_conditionspeed(threading.Thread, threading.Condition())
    print '\n\t######## testing multiprocessing.Condition\n'
    test_conditionspeed(multiprocessing.Process, multiprocessing.Condition())
    print '\n\t######## testing condition managed by a server process\n'
    test_conditionspeed(multiprocessing.Process, manager.Condition())

    gc.enable()

if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

16.7 mmap — Memory-mapped file support

Memory-mapped file objects behave like both strings and like file objects. Unlike normal string objects, however, these are mutable. You can use mmap objects in most places where strings are expected; for example, you can use the re module to search through a memory-mapped file. Since they're mutable, you can change a single character by doing obj[index] = 'a', or change a substring by assigning to a slice: obj[i1:i2] = '...'. You can also read and write data starting at the current file position, and seek() through the file to different positions.

A memory-mapped file is created by the mmap constructor, which is different on Unix and on Windows. In either case you must provide a file descriptor for a file opened for update. If you wish to map an existing Python file object, use its fileno() method to obtain the correct value for the fileno parameter. Otherwise,

you can open the file using the os.open() function, which returns a file descriptor directly (the file still needs to be closed when done).

Note: If you want to create a memory-mapping for a writable, buffered file, you should flush() the file first. This is necessary to ensure that local modifications to the buffers are actually available to the mapping.

For both the Unix and Windows versions of the constructor, access may be specified as an optional keyword parameter. access accepts one of three values: ACCESS_READ, ACCESS_WRITE, or ACCESS_COPY to specify read-only, write-through or copy-on-write memory respectively. access can be used on both Unix and Windows. If access is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an ACCESS_READ memory map raises a TypeError exception. Assignment to an ACCESS_WRITE memory map affects both memory and the underlying file. Assignment to an ACCESS_COPY memory map affects memory but does not update the underlying file.

Changed in version 2.5: To map anonymous memory, -1 should be passed as the fileno along with the length.

Changed in version 2.6: mmap.mmap has formerly been a factory function creating mmap objects. Now mmap.mmap is the class itself.

class mmap.mmap(fileno, length[, tagname[, access[, offset]]])
(Windows version) Maps length bytes from the file specified by the file handle fileno, and creates a mmap object. If length is larger than the current size of the file, the file is extended to contain length bytes. If length is 0, the maximum length of the map is the current size of the file, except that if the file is empty Windows raises an exception (you cannot create an empty mapping on Windows).

tagname, if specified and not None, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened, otherwise a new tag of this name is created. If this parameter is omitted or None, the mapping is created without a name. Avoiding the use of the tag parameter will assist in keeping your code portable between Unix and Windows.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. offset defaults to 0. offset must be a multiple of the ALLOCATIONGRANULARITY.

class mmap.mmap(fileno, length[, flags[, prot[, access[, offset]]]])
(Unix version) Maps length bytes from the file specified by the file descriptor fileno, and returns a mmap object. If length is 0, the maximum length of the map will be the current size of the file when mmap is called.

flags specifies the nature of the mapping. MAP_PRIVATE creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and MAP_SHARED creates a mapping that's shared with all other processes mapping the same areas of the file. The default value is MAP_SHARED.

prot, if specified, gives the desired memory protection; the two most useful values are PROT_READ and PROT_WRITE, to specify that the pages may be read or written. prot defaults to PROT_READ | PROT_WRITE.

access may be specified in lieu of flags and prot as an optional keyword parameter. It is an error to specify both flags, prot and access. See the description of access above for information on how to use this parameter.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. offset defaults to 0. offset must be a multiple of ALLOCATIONGRANULARITY which is equal to PAGESIZE on Unix systems.

To ensure validity of the created memory mapping the file specified by the descriptor fileno is internally automatically synchronized with physical backing store on Mac OS X and OpenVMS.

This example shows a simple way of using mmap:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write("Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print mm.readline()  # prints "Hello Python!"
    # read content via slice notation
    print mm[:5]  # prints "Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = " world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print mm.readline()  # prints "Hello  world!"
    # close the map
    mm.close()
```

The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write("Hello world!")

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print mm.readline()

    mm.close()
```

Memory-mapped file objects support the following methods:

close()
: Closes the mmap. Subsequent calls to other methods of the object will result in a ValueError exception being raised. This will not close the open file.

find(string[, start[, end]])
: Returns the lowest index in the object where the substring string is found, such that string is contained in the range [start, end]. Optional arguments start and end are interpreted as in slice notation. Returns -1 on failure.

flush([offset, size])
: Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If offset and size are specified, only changes to the given range of bytes will be flushed to disk; otherwise,

the whole extent of the mapping is flushed. offset must be a multiple of the PAGESIZE or ALLOCATIONGRANULARITY.

(Windows version) A nonzero value returned indicates success; zero indicates failure.

(Unix version) A zero value is returned to indicate success. An exception is raised when the call failed.

move(dest, src, count)
Copy the count bytes starting at offset src to the destination index dest. If the mmap was created with ACCESS_READ, then calls to move will raise a TypeError exception.

read(num)
Return a string containing up to num bytes starting from the current file position; the file position is updated to point after the bytes that were returned.

read_byte()
Returns a string of length 1 containing the character at the current file position, and advances the file position by 1.

readline()
Returns a single line, starting at the current file position and up to the next newline.

resize(newsize)
Resizes the map and the underlying file, if any. If the mmap was created with ACCESS_READ or ACCESS_COPY, resizing the map will raise a TypeError exception.

rfind(string[, start[, end]])
Returns the highest index in the object where the substring string is found, such that string is contained in the range [start, end]. Optional arguments start and end are interpreted as in slice notation. Returns -1 on failure.

seek(pos[, whence])
Set the file's current position. whence argument is optional and defaults to os.SEEK_SET or 0 (absolute file positioning); other values are os.SEEK_CUR or 1 (seek relative to the current position) and os.SEEK_END or 2 (seek relative to the file's end).

size()
Return the length of the file, which can be larger than the size of the memory-mapped area.

tell()
Returns the current position of the file pointer.

write(string)
Write the bytes in string into memory at the current position of the file pointer; the file position is updated to point after the bytes that were written. If the mmap was created with ACCESS_READ, then writing to it will raise a TypeError exception.

write_byte(byte)
Write the single-character string byte into memory at the current position of the file pointer; the file position is advanced by 1. If the mmap was created with ACCESS_READ, then writing to it will raise a TypeError exception.

16.8 readline — GNU readline interface

The readline module defines a number of functions to facilitate completion and reading/writing of history files from the Python interpreter. This module can be used directly, or via the rlcompleter module, which supports completion of Python identifiers at the interactive prompt. Settings made using this module affect

the behaviour of both the interpreter's interactive prompt and the prompts offered by the raw_input() and input() built-in functions.

Note: The underlying Readline library API may be implemented by the libedit library instead of GNU readline. On MacOS X the readline module detects which library is being used at run time.

The configuration file for libedit is different from that of GNU readline. If you programmatically load configuration strings you can check for the text "libedit" in readline.__doc__ to differentiate between GNU readline and libedit.

Readline keybindings may be configured via an initialization file, typically .inputrc in your home directory. See Readline Init File in the GNU Readline manual for information about the format and allowable constructs of that file, and the capabilities of the Readline library in general.

16.8.1 Init file

The following functions relate to the init file and user configuration:

readline.parse_and_bind(string)
: Execute the init line provided in the string argument. This calls rl_parse_and_bind() in the underlying library.

readline.read_init_file([filename])
: Execute a readline initialization file. The default filename is the last filename used. This calls rl_read_init_file() in the underlying library.

16.8.2 Line buffer

The following functions operate on the line buffer:

readline.get_line_buffer()
: Return the current contents of the line buffer (rl_line_buffer in the underlying library).

readline.insert_text(string)
: Insert text into the line buffer at the cursor position. This calls rl_insert_text() in the underlying library, but ignores the return value.

readline.redisplay()
: Change what's displayed on the screen to reflect the current contents of the line buffer. This calls rl_redisplay() in the underlying library.

16.8.3 History file

The following functions operate on a history file:

readline.read_history_file([filename])
: Load a readline history file, and append it to the history list. The default filename is ~/.history. This calls read_history() in the underlying library.

readline.write_history_file([filename])
: Save the history list to a readline history file, overwriting any existing file. The default filename is ~/.history. This calls write_history() in the underlying library.

readline.get_history_length()

readline.set_history_length(length)
Set or return the desired number of lines to save in the history file. The write_history_file() function uses this value to truncate the history file, by calling history_truncate_file() in the underlying library. Negative values imply unlimited history file size.

16.8.4 History list

The following functions operate on a global history list:

readline.clear_history()
Clear the current history. This calls clear_history() in the underlying library. The Python function only exists if Python was compiled for a version of the library that supports it.

New in version 2.4.

readline.get_current_history_length()
Return the number of items currently in the history. (This is different from get_history_length(), which returns the maximum number of lines that will be written to a history file.)

New in version 2.3.

readline.get_history_item(index)
Return the current contents of history item at index. The item index is one-based. This calls history_get() in the underlying library.

New in version 2.3.

readline.remove_history_item(pos)
Remove history item specified by its position from the history. The position is zero-based. This calls remove_history() in the underlying library.

New in version 2.4.

readline.replace_history_item(pos, line)
Replace history item specified by its position with line. The position is zero-based. This calls replace_history_entry() in the underlying library.

New in version 2.4.

readline.add_history(line)
Append line to the history buffer, as if it was the last line typed. This calls add_history() in the underlying library.

16.8.5 Startup hooks

New in version 2.3.

readline.set_startup_hook([function])
Set or remove the function invoked by the rl_startup_hook callback of the underlying library. If function is specified, it will be used as the new hook function; if omitted or None, any function already installed is removed. The hook is called with no arguments just before readline prints the first prompt.

readline.set_pre_input_hook([function])
Set or remove the function invoked by the rl_pre_input_hook callback of the underlying library. If function is specified, it will be used as the new hook function; if omitted or None, any function already installed is removed. The hook is called with no arguments after the first prompt has been printed and just before readline starts reading input characters. This function only exists if Python was compiled for a version of the library that supports it.

16.8.6 Completion

The following functions relate to implementing a custom word completion function. This is typically operated by the Tab key, and can suggest and automatically complete a word being typed. By default, Readline is set up to be used by rlcompleter to complete Python identifiers for the interactive interpreter. If the readline module is to be used with a custom completer, a different set of word delimiters should be set.

readline.set_completer([function])
: Set or remove the completer function. If function is specified, it will be used as the new completer function; if omitted or None, any completer function already installed is removed. The completer function is called as function(text, state), for state in 0, 1, 2, ..., until it returns a non-string value. It should return the next possible completion starting with text.

  The installed completer function is invoked by the entry_func callback passed to rl_completion_matches() in the underlying library. The text string comes from the first parameter to the rl_attempted_completion_function callback of the underlying library.

readline.get_completer()
: Get the completer function, or None if no completer function has been set.

  New in version 2.3.

readline.get_completion_type()
: Get the type of completion being attempted. This returns the rl_completion_type variable in the underlying library as an integer.

  New in version 2.6.

readline.get_begidx()
readline.get_endidx()
: Get the beginning or ending index of the completion scope. These indexes are the start and end arguments passed to the rl_attempted_completion_function callback of the underlying library.

readline.set_completer_delims(string)
readline.get_completer_delims()
: Set or get the word delimiters for completion. These determine the start of the word to be considered for completion (the completion scope). These functions access the rl_completer_word_break_characters variable in the underlying library.

readline.set_completion_display_matches_hook([function])
: Set or remove the completion display function. If function is specified, it will be used as the new completion display function; if omitted or None, any completion display function already installed is removed. This sets or clears the rl_completion_display_matches_hook callback in the underlying library. The completion display function is called as function(substitution, [matches], longest_match_length) once each time matches need to be displayed.

  New in version 2.6.

16.8.7 Example

The following example demonstrates how to use the readline module's history reading and writing functions to automatically load and save a history file named .pyhist from the user's home directory. The code below would normally be executed automatically during interactive sessions from the user's PYTHONSTARTUP file.

```
import os
import readline
```

(continues on next page)

(continued from previous page)

```
histfile = os.path.join(os.path.expanduser("~"), ".pyhist")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except IOError:
    pass
import atexit
atexit.register(readline.write_history_file, histfile)
del os, histfile
```

The following example extends the code.InteractiveConsole class to support history save/restore.

```
import code
import readline
import atexit
import os

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except IOError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

16.9 rlcompleter — Completion function for GNU readline

Source code: Lib/rlcompleter.py

The rlcompleter module defines a completion function suitable for the readline module by completing valid Python identifiers and keywords.

When this module is imported on a Unix platform with the readline module available, an instance of the Completer class is automatically created and its complete() method is set as the readline completer.

Example:

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
```

(continues on next page)

(continued from previous page)

```
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__         readline.insert_text(      readline.set_completer(
readline.__name__         readline.parse_and_bind(
>>> readline.
```

The rlcompleter module is designed for use with Python's interactive mode. A user can add the following lines to his or her initialization file (identified by the PYTHONSTARTUP environment variable) to get automatic Tab completion:

```
try:
    import readline
except ImportError:
    print "Module readline not available."
else:
    import rlcompleter
    readline.parse_and_bind("tab: complete")
```

On platforms without readline, the Completer class defined by this module can still be used for custom purposes.

16.9.1 Completer Objects

Completer objects have the following method:

Completer.complete(text, state)

Return the stateth completion for text.

If called for text that doesn't include a period character (`'.'`), it will complete from names currently defined in __main__, __builtin__ and keywords (as defined by the keyword module).

If called for a dotted name, it will try to evaluate anything without obvious side-effects (functions will not be evaluated, but it can generate calls to __getattr__()) up to the last part, and find matches for the rest via the dir() function. Any exception raised during the evaluation of the expression is caught, silenced and None is returned.

CHAPTER

SEVENTEEN

INTERPROCESS COMMUNICATION AND NETWORKING

The modules described in this chapter provide mechanisms for different processes to communicate.

Some modules only work for two processes that are on the same machine, e.g. signal and subprocess. Other modules support networking protocols that two or more processes can use to communicate across machines.

The list of modules described in this chapter is:

17.1 subprocess — Subprocess management

New in version 2.4.

The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

```
os.system
os.spawn*
os.popen*
popen2.*
commands.*
```

Information about how this module can be used to replace the older functions can be found in the subprocess-replacements section.

See also:

POSIX users (Linux, BSD, etc.) are strongly encouraged to install and use the much more recent subprocess32 module instead of the version included with python 2.7. It is a drop in replacement with better behavior in many situations.

PEP 324 – PEP proposing the subprocess module

17.1.1 Using the subprocess Module

The recommended way to launch subprocesses is to use the following convenience functions. For more advanced use cases when these do not meet your needs, use the underlying Popen interface.

subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False)

Run the command described by args. Wait for command to complete, then return the returncode attribute.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the slightly odd notation in the abbreviated signature). The full function signature

is the same as that of the Popen constructor - this functions passes all supplied arguments directly through to that interface.

Examples:

```
>>> subprocess.call(["ls", "-l"])
0

>>> subprocess.call("exit 1", shell=True)
1
```

> Warning: Using shell=True can be a security hazard. See the warning under Frequently Used Arguments for details.

Note: Do not use stdout=PIPE or stderr=PIPE with this function as that can deadlock based on the child process output volume. Use Popen with the communicate() method when you need pipes.

---

subprocess.check_call(args, *, stdin=None, stdout=None, stderr=None, shell=False)

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise CalledProcessError. The CalledProcessError object will have the return code in the returncode attribute.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the slightly odd notation in the abbreviated signature). The full function signature is the same as that of the Popen constructor - this functions passes all supplied arguments directly through to that interface.

Examples:

```
>>> subprocess.check_call(["ls", "-l"])
0

>>> subprocess.check_call("exit 1", shell=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1
```

New in version 2.5.

> Warning: Using shell=True can be a security hazard. See the warning under Frequently Used Arguments for details.

Note: Do not use stdout=PIPE or stderr=PIPE with this function as that can deadlock based on the child process output volume. Use Popen with the communicate() method when you need pipes.

---

subprocess.check_output(args, *, stdin=None, stderr=None, shell=False, universal_newlines=False)

Run command with arguments and return its output as a byte string.

If the return code was non-zero it raises a CalledProcessError. The CalledProcessError object will have the return code in the returncode attribute and any output in the output attribute.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the slightly odd notation in the abbreviated signature). The full function signature is largely the same as that of the Popen constructor, except that stdout is not permitted as it is used internally. All other supplied arguments are passed directly through to the Popen constructor.

Examples:

```
>>> subprocess.check_output(["echo", "Hello World!"])
'Hello World!\n'

>>> subprocess.check_output("exit 1", shell=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1
```

To also capture standard error in the result, use stderr=subprocess.STDOUT:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

New in version 2.7.

> Warning: Using shell=True can be a security hazard. See the warning under Frequently Used Arguments for details.

Note: Do not use stderr=PIPE with this function as that can deadlock based on the child process error volume. Use Popen with the communicate() method when you need a stderr pipe.

---

subprocess.PIPE
: Special value that can be used as the stdin, stdout or stderr argument to Popen and indicates that a pipe to the standard stream should be opened.

subprocess.STDOUT
: Special value that can be used as the stderr argument to Popen and indicates that standard error should go into the same handle as standard output.

exception subprocess.CalledProcessError
: Exception raised when a process run by check_call() or check_output() returns a non-zero exit status.

  returncode
  : Exit status of the child process.

  cmd
  : Command that was used to spawn the child process.

  output
  : Output of the child process if this exception is raised by check_output(). Otherwise, None.

Frequently Used Arguments

To support a wide variety of use cases, the Popen constructor (and the convenience functions) accept a large number of optional arguments. For most typical use cases, many of these arguments can be safely left at their default values. The arguments that are most commonly needed are:

args is required for all calls and should be a string, or a sequence of program arguments. Providing a sequence of arguments is generally preferred, as it allows the module to take care of any required escaping and quoting of arguments (e.g. to permit spaces in file names). If passing a single string, either shell must be True (see below) or else the string must simply name the program to be executed without specifying any arguments.

stdin, stdout and stderr specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are PIPE, an existing file descriptor (a positive integer), an existing file object, and None. PIPE indicates that a new pipe to the child should be created. With the default settings of None, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, stderr can be STDOUT, which indicates that the stderr data from the child process should be captured into the same file handle as for stdout.

When stdout or stderr are pipes and universal_newlines is True then all line endings will be converted to `'\n'` as described for the universal newlines `'U'` mode argument to open().

If shell is True, the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of ~ to a user's home directory. However, note that Python itself offers implementations of many shell-like features (in particular, glob, fnmatch, os.walk(), os.path.expandvars(), os.path.expanduser(), and shutil).

Warning: Executing shell commands that incorporate unsanitized input from an untrusted source makes a program vulnerable to shell injection, a serious security flaw which can result in arbitrary command execution. For this reason, the use of shell=True is strongly discouraged in cases where the command string is constructed from external input:

```
>>> from subprocess import call
>>> filename = input("What file would you like to display?\n")
What file would you like to display?
non_existent; rm -rf / #
>>> call("cat " + filename, shell=True) # Uh-oh. This will end badly...
```

shell=False disables all shell based features, but does not suffer from this vulnerability; see the Note in the Popen constructor documentation for helpful hints in getting shell=False to work.

When using shell=True, pipes.quote() can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

These options, along with all of the other options, are described in more detail in the Popen constructor documentation.

Popen Constructor

The underlying process creation and management in this module is handled by the Popen class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions.

class subprocess.Popen(args, bufsize=0, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=False, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0)

Execute a child program in a new process. On Unix, the class uses os.execvp()-like behavior to execute the child program. On Windows, the class uses the Windows CreateProcess() function. The arguments to Popen are as follows.

args should be a sequence of program arguments or else a single string. By default, the program to execute is the first item in args if args is a sequence. If args is a string, the interpretation is platform-dependent and described below. See the shell and executable arguments for additional differences from the default behavior. Unless otherwise stated, it is recommended to pass args as a sequence.

On Unix, if args is a string, the string is interpreted as the name or path of the program to execute. However, this can only be done if not passing arguments to the program.

Note: shlex.split() can be useful when determining the correct tokenization for args, especially in complex cases:

```
>>> import shlex, subprocess
>>> command_line = raw_input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print args
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', "echo '$MONEY'"]
>>> p = subprocess.Popen(args) # Success!
```

Note in particular that options (such as -input) and arguments (such as eggs.txt) that are separated by whitespace in the shell go in separate list elements, while arguments that need quoting or backslash escaping when used in the shell (such as filenames containing spaces or the echo command shown above) are single list elements.

On Windows, if args is a sequence, it will be converted to a string in a manner described in Converting an argument sequence to a string on Windows. This is because the underlying CreateProcess() operates on strings.

The shell argument (which defaults to False) specifies whether to use the shell as the program to execute. If shell is True, it is recommended to pass args as a string rather than as a sequence.

On Unix with shell=True, the shell defaults to /bin/sh. If args is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If args is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, Popen does the equivalent of:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

On Windows with shell=True, the COMSPEC environment variable specifies the default shell. The only time you need to specify shell=True on Windows is when the command you wish to execute is built into the shell (e.g. dir or copy). You do not need shell=True to run a batch file or console-based executable.

Warning: Passing shell=True can be a security hazard if combined with untrusted input. See the warning under Frequently Used Arguments for details.

bufsize, if given, has the same meaning as the corresponding argument to the built-in open() function: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size. A negative bufsize means to use the system default, which usually means fully buffered. The default value for bufsize is 0 (unbuffered).

Note: If you experience performance issues, it is recommended that you try to enable buffering by setting bufsize to either -1 or a large enough positive value (such as 4096).

The executable argument specifies a replacement program to execute. It is very seldom needed. When shell=False, executable replaces the program to execute specified by args. However, the original args is still passed to the program. Most programs treat the program specified by args as the command name, which can then be different from the program actually executed. On Unix, the args name becomes the display name for the executable in utilities such as ps. If shell=True, on Unix the executable argument specifies a replacement shell for the default /bin/sh.

stdin, stdout and stderr specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are PIPE, an existing file descriptor (a positive integer), an existing file object, and None. PIPE indicates that a new pipe to the child should be created. With the default settings of None, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, stderr can be STDOUT, which indicates that the stderr data from the child process should be captured into the same file handle as for stdout.

If preexec_fn is set to a callable object, this object will be called in the child process just before the child is executed. (Unix only)

If close_fds is true, all file descriptors except 0, 1 and 2 will be closed before the child process is executed. (Unix only). Or, on Windows, if close_fds is true then no handles will be inherited by the child process. Note that on Windows, you cannot set close_fds to true and also redirect the standard handles by setting stdin, stdout or stderr.

If cwd is not None, the child's current directory will be changed to cwd before it is executed. Note that this directory is not considered when searching the executable, so you can't specify the program's path relative to cwd.

If env is not None, it must be a mapping that defines the environment variables for the new process; these are used instead of inheriting the current process' environment, which is the default behavior.

Note: If specified, env must provide any variables required for the program to execute. On Windows, in order to run a side-by-side assembly the specified env must include a valid SystemRoot.

If universal_newlines is True, the file objects stdout and stderr are opened as text files in universal newlines mode. Lines may be terminated by any of `'\n'`, the Unix end-of-line convention, `'\r'`, the old Macintosh convention or `'\r\n'`, the Windows convention. All of these external representations are seen as `'\n'` by the Python program.

Note: This feature is only available if Python is built with universal newline support (the default). Also, the newlines attribute of the file objects stdout, stdin and stderr are not updated by the communicate() method.

If given, startupinfo will be a STARTUPINFO object, which is passed to the underlying CreateProcess function. creationflags, if given, can be CREATE_NEW_CONSOLE or CREATE_NEW_PROCESS_GROUP. (Windows only)

Exceptions

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent. Additionally, the exception object will have one extra attribute called child_traceback, which is a string containing traceback information from the child's point of view.

The most common exception raised is OSError. This occurs, for example, when trying to execute a non-existent file. Applications should prepare for OSError exceptions.

A ValueError will be raised if Popen is called with invalid arguments.

check_call() and check_output() will raise CalledProcessError if the called process returns a non-zero return code.

Security

Unlike some other popen functions, this implementation will never call a system shell implicitly. This means that all characters, including shell metacharacters, can safely be passed to child processes. Obviously, if the shell is invoked explicitly, then it is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately.

17.1.2 Popen Objects

Instances of the Popen class have the following methods:

Popen.poll()
: Check if child process has terminated. Set and return returncode attribute.

Popen.wait()
: Wait for child process to terminate. Set and return returncode attribute.

> Warning: This will deadlock when using stdout=PIPE and/or stderr=PIPE and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use communicate() to avoid that.

Popen.communicate(input=None)
: Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional input argument should be a string to be sent to the child process, or None, if no data should be sent to the child.

  communicate() returns a tuple (stdoutdata, stderrdata).

  Note that if you want to send data to the process's stdin, you need to create the Popen object with stdin=PIPE. Similarly, to get anything other than None in the result tuple, you need to give stdout=PIPE and/or stderr=PIPE too.

  Note: The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

Popen.send_signal(signal)
: Sends the signal signal to the child.

Note: On Windows, SIGTERM is an alias for terminate(). CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a creationflags parameter which includes CREATE_NEW_PROCESS_GROUP.

New in version 2.6.

Popen.terminate()
Stop the child. On Posix OSs the method sends SIGTERM to the child. On Windows the Win32 API function TerminateProcess() is called to stop the child.

New in version 2.6.

Popen.kill()
Kills the child. On Posix OSs the function sends SIGKILL to the child. On Windows kill() is an alias for terminate().

New in version 2.6.

The following attributes are also available:

Warning: Use communicate() rather than .stdin.write, .stdout.read or .stderr.read to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

Popen.stdin
If the stdin argument was PIPE, this attribute is a file object that provides input to the child process. Otherwise, it is None.

Popen.stdout
If the stdout argument was PIPE, this attribute is a file object that provides output from the child process. Otherwise, it is None.

Popen.stderr
If the stderr argument was PIPE, this attribute is a file object that provides error output from the child process. Otherwise, it is None.

Popen.pid
The process ID of the child process.

Note that if you set the shell argument to True, this is the process ID of the spawned shell.

Popen.returncode
The child return code, set by poll() and wait() (and indirectly by communicate()). A None value indicates that the process hasn't terminated yet.

A negative value -N indicates that the child was terminated by signal N (Unix only).

17.1.3 Windows Popen Helpers

The STARTUPINFO class and following constants are only available on Windows.

class subprocess.STARTUPINFO
Partial support of the Windows STARTUPINFO structure is used for Popen creation.

dwFlags
A bit field that determines whether certain STARTUPINFO attributes are used when the process creates a window.

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

hStdInput
: If dwFlags specifies STARTF_USESTDHANDLES, this attribute is the standard input handle for the process. If STARTF_USESTDHANDLES is not specified, the default for standard input is the keyboard buffer.

hStdOutput
: If dwFlags specifies STARTF_USESTDHANDLES, this attribute is the standard output handle for the process. Otherwise, this attribute is ignored and the default for standard output is the console window's buffer.

hStdError
: If dwFlags specifies STARTF_USESTDHANDLES, this attribute is the standard error handle for the process. Otherwise, this attribute is ignored and the default for standard error is the console window's buffer.

wShowWindow
: If dwFlags specifies STARTF_USESHOWWINDOW, this attribute can be any of the values that can be specified in the nCmdShow parameter for the ShowWindow function, except for SW_SHOWDEFAULT. Otherwise, this attribute is ignored.

  SW_HIDE is provided for this attribute. It is used when Popen is called with shell=True.

Constants

The subprocess module exposes the following constants.

subprocess.STD_INPUT_HANDLE
: The standard input device. Initially, this is the console input buffer, CONIN$.

subprocess.STD_OUTPUT_HANDLE
: The standard output device. Initially, this is the active console screen buffer, CONOUT$.

subprocess.STD_ERROR_HANDLE
: The standard error device. Initially, this is the active console screen buffer, CONOUT$.

subprocess.SW_HIDE
: Hides the window. Another window will be activated.

subprocess.STARTF_USESTDHANDLES
: Specifies that the STARTUPINFO.hStdInput, STARTUPINFO.hStdOutput, and STARTUPINFO.hStdError attributes contain additional information.

subprocess.STARTF_USESHOWWINDOW
: Specifies that the STARTUPINFO.wShowWindow attribute contains additional information.

subprocess.CREATE_NEW_CONSOLE
: The new process has a new console, instead of inheriting its parent's console (the default).

  This flag is always set when Popen is created with shell=True.

subprocess.CREATE_NEW_PROCESS_GROUP
: A Popen creationflags parameter to specify that a new process group will be created. This flag is necessary for using os.kill() on the subprocess.

  This flag is ignored if CREATE_NEW_CONSOLE is specified.

17.1.4 Replacing Older Functions with the subprocess Module

In this section, "a becomes b" means that b can be used as a replacement for a.

Note: All "a" functions in this section fail (more or less) silently if the executed program cannot be found; the "b" replacements raise OSError instead.

In addition, the replacements using check_output() will fail with a CalledProcessError if the requested operation produces a non-zero return code. The output is still available as the output attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from the subprocess module.

Replacing /bin/sh shell backquote

```
output=`mycmd myarg`
```

becomes:

```
output = check_output(["mycmd", "myarg"])
```

Replacing shell pipeline

```
output=`dmesg | grep hda`
```

becomes:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

The p1.stdout.close() call after starting the p2 is important in order for p1 to receive a SIGPIPE if p2 exits before p1.

Alternatively, for trusted input, the shell's own pipeline support may still be used directly:

```
output=`dmesg | grep hda`
```

becomes:

```
output=check_output("dmesg | grep hda", shell=True)
```

Replacing os.system()

```
status = os.system("mycmd" + " myarg")
# becomes
status = subprocess.call("mycmd" + " myarg", shell=True)
```

Notes:

- Calling the program through the shell is usually not required.

A more realistic example would look like this:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print >>sys.stderr, "Child was terminated by signal", -retcode
    else:
        print >>sys.stderr, "Child returned", retcode
except OSError as e:
    print >>sys.stderr, "Execution failed:", e
```

Replacing the os.spawn family

P_NOWAIT example:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT example:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Vector example:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Environment example:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

Replacing os.popen(), os.popen2(), os.popen3()

```
pipe = os.popen("cmd", 'r', bufsize)
==>
pipe = Popen("cmd", shell=True, bufsize=bufsize, stdout=PIPE).stdout
```

```
pipe = os.popen("cmd", 'w', bufsize)
==>
pipe = Popen("cmd", shell=True, bufsize=bufsize, stdin=PIPE).stdin
```

```
(child_stdin, child_stdout) = os.popen2("cmd", mode, bufsize)
==>
p = Popen("cmd", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3("cmd", mode, bufsize)
==>
p = Popen("cmd", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4("cmd", mode,
                                                   bufsize)
==>
p = Popen("cmd", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

On Unix, os.popen2, os.popen3 and os.popen4 also accept a sequence as the command to execute, in which case arguments will be passed directly to the program without shell intervention. This usage can be replaced as follows:

```
(child_stdin, child_stdout) = os.popen2(["/bin/ls", "-l"], mode,
                                        bufsize)
==>
p = Popen(["/bin/ls", "-l"], bufsize=bufsize, stdin=PIPE, stdout=PIPE)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

Return code handling translates as follows:

```
pipe = os.popen("cmd", 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print "There were some errors"
==>
process = Popen("cmd", shell=True, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print "There were some errors"
```

Replacing functions from the popen2 module

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

On Unix, popen2 also accepts a sequence as the command to execute, in which case arguments will be passed directly to the program without shell intervention. This usage can be replaced as follows:

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize,
                                            mode)
==>
```

(continues on next page)

(continued from previous page)

```
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

popen2.Popen3 and popen2.Popen4 basically work as subprocess.Popen, except that:

- Popen raises an exception if the execution fails.
- the capturestderr argument is replaced with the stderr argument.
- stdin=PIPE and stdout=PIPE must be specified.
- popen2 closes all file descriptors by default, but you have to specify close_fds=True with Popen.

17.1.5 Notes

Converting an argument sequence to a string on Windows

On Windows, an args sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.
2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.

17.2 socket — Low-level networking interface

This module provides access to the BSD socket interface. It is available on all modern Unix systems, Windows, Mac OS X, BeOS, OS/2, and probably additional platforms.

Note: Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

For an introduction to socket programming (in C), see the following papers: An Introductory 4.3BSD Interprocess Communication Tutorial, by Stuart Sechrest and An Advanced 4.3BSD Interprocess Communication Tutorial, by Samuel J. Leffler et al, both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to RFC 3493 titled Basic Socket Interface Extensions for IPv6.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style: the socket() function returns a socket object whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with read() and write() operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

Socket addresses are represented as follows: A single string is used for the AF_UNIX address family. A pair (host, port) is used for the AF_INET address family, where host is a string representing either a hostname in Internet domain notation like `'daring.cwi.nl'` or an IPv4 address like `'100.50.200.5'`, and port is an integer. For AF_INET6 address family, a four-tuple (host, port, flowinfo, scopeid) is used, where flowinfo and scopeid represents sin6_flowinfo and sin6_scope_id member in struct sockaddr_in6 in C. For socket module methods, flowinfo and scopeid can be omitted just for backward compatibility. Note, however, omission of scopeid can cause problems in manipulating scoped IPv6 addresses. Other address families are currently not supported. The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created.

For IPv4 addresses, two special forms are accepted instead of a host address: the empty string represents INADDR_ANY, and the string `'<broadcast>'` represents INADDR_BROADCAST. The behavior is not available for IPv6 for backward compatibility, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.

If you use a hostname in the host portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in host portion.

New in version 2.5: AF_NETLINK sockets are represented as pairs pid, groups.

New in version 2.6: Linux-only support for TIPC is also available using the AF_TIPC address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is (addr_type, v1, v2, v3 [, scope]), where:

- addr_type is one of TIPC_ADDR_NAMESEQ, TIPC_ADDR_NAME, or TIPC_ADDR_ID.
- scope is one of TIPC_ZONE_SCOPE, TIPC_CLUSTER_SCOPE, and TIPC_NODE_SCOPE.
- If addr_type is TIPC_ADDR_NAME, then v1 is the server type, v2 is the port identifier, and v3 should be 0.

  If addr_type is TIPC_ADDR_NAMESEQ, then v1 is the server type, v2 is the lower port number, and v3 is the upper port number.

  If addr_type is TIPC_ADDR_ID, then v1 is the node, v2 is the reference, and v3 should be set to 0.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; errors related to socket or address semantics raise the error socket.error.

Non-blocking mode is supported through setblocking(). A generalization of this based on timeouts is supported through settimeout().

The module socket exports the following constants and functions:

exception socket.error
: This exception is raised for socket-related errors. The accompanying value is either a string telling what went wrong or a pair (errno, string) representing an error returned by a system call, similar to the value accompanying os.error. See the module errno, which contains names for the error codes defined by the underlying operating system.

  Changed in version 2.6: socket.error is now a child class of IOError.

exception socket.herror
: This exception is raised for address-related errors, i.e. for functions that use h_errno in the C API, including gethostbyname_ex() and gethostbyaddr().

  The accompanying value is a pair (h_errno, string) representing an error returned by a library call. string represents the description of h_errno, as returned by the hstrerror() C function.

exception socket.gaierror
: This exception is raised for address-related errors, for getaddrinfo() and getnameinfo(). The accompanying value is a pair (error, string) representing an error returned by a library call. string represents the description of error, as returned by the gai_strerror() C function. The error value will match one of the EAI_* constants defined in this module.

exception socket.timeout
: This exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to settimeout(). The accompanying value is a string whose value is currently always "timed out".

  New in version 2.3.

socket.AF_UNIX
socket.AF_INET
socket.AF_INET6
: These constants represent the address (and protocol) families, used for the first argument to socket(). If the AF_UNIX constant is not defined then this protocol is unsupported.

socket.SOCK_STREAM
socket.SOCK_DGRAM
socket.SOCK_RAW
socket.SOCK_RDM
socket.SOCK_SEQPACKET
: These constants represent the socket types, used for the second argument to socket(). (Only SOCK_STREAM and SOCK_DGRAM appear to be generally useful.)

SO_*
socket.SOMAXCONN
MSG_*
SOL_*
IPPROTO_*
IPPORT_*
INADDR_*
IP_*
IPV6_*
EAI_*
AI_*
NI_*
TCP_*
: Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the setsockopt() and getsockopt() methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

SIO_*
RCVALL_*
: Constants for Windows' WSAIoctl(). The constants are used as arguments to the ioctl() method of socket objects.

  New in version 2.6.

TIPC_*
: TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

  New in version 2.6.

socket.has_ipv6
: This constant contains a boolean value which indicates if IPv6 is supported on this platform.

New in version 2.3.

socket.create_connection(address[, timeout[, source_address]])

Connect to a TCP service listening on the Internet address (a 2-tuple (host, port)), and return the socket object. This is a higher-level function than socket.connect(): if host is a non-numeric hostname, it will try to resolve it for both AF_INET and AF_INET6, and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

Passing the optional timeout parameter will set the timeout on the socket instance before attempting to connect. If no timeout is supplied, the global default timeout setting returned by getdefaulttimeout() is used.

If supplied, source_address must be a 2-tuple (host, port) for the socket to bind to as its source address before connecting. If host or port are '' or 0 respectively the OS default behavior will be used.

New in version 2.6.

Changed in version 2.7: source_address was added.

socket.getaddrinfo(host, port[, family[, socktype[, proto[, flags]]]])

Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. host is a domain name, a string representation of an IPv4/v6 address or None. port is a string service name such as 'http', a numeric port number or None. By passing None as the value of host and port, you can pass NULL to the underlying C API.

The family, socktype and proto arguments can be optionally specified in order to narrow the list of addresses returned. By default, their value is 0, meaning that the full range of results is selected. The flags argument can be one or several of the AI_* constants, and will influence how results are computed and returned. Its default value is 0. For example, AI_NUMERICHOST will disable domain name resolution and will raise an error if host is a domain name.

The function returns a list of 5-tuples with the following structure:

(family, socktype, proto, canonname, sockaddr)

In these tuples, family, socktype, proto are all integers and are meant to be passed to the socket() function. canonname will be a string representing the canonical name of the host if AI_CANONNAME is part of the flags argument; else canonname will be empty. sockaddr is a tuple describing a socket address, whose format depends on the returned family (a (address, port) 2-tuple for AF_INET, a (address, port, flow info, scope id) 4-tuple for AF_INET6), and is meant to be passed to the socket.connect() method.

The following example fetches address information for a hypothetical TCP connection to example.org on port 80 (results may differ on your system if IPv6 isn't enabled):

```
>>> socket.getaddrinfo("example.org", 80, 0, 0, socket.IPPROTO_TCP)
[(10, 1, 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (2, 1, 6, '', ('93.184.216.34', 80))]
```

New in version 2.2.

socket.getfqdn([name])

Return a fully qualified domain name for name. If name is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by gethostbyaddr() is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available, the hostname as returned by gethostname() is returned.

New in version 2.0.

socket.gethostbyname(hostname)
Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as '100.50.200.5'. If the host name is an IPv4 address itself it is returned unchanged. See gethostbyname_ex() for a more complete interface. gethostbyname() does not support IPv6 name resolution, and getaddrinfo() should be used instead for IPv4/v6 dual stack support.

socket.gethostbyname_ex(hostname)
Translate a host name to IPv4 address format, extended interface. Return a triple (hostname, aliaslist, ipaddrlist) where hostname is the primary host name responding to the given ip_address, aliaslist is a (possibly empty) list of alternative host names for the same address, and ipaddrlist is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). gethostbyname_ex() does not support IPv6 name resolution, and getaddrinfo() should be used instead for IPv4/v6 dual stack support.

socket.gethostname()
Return a string containing the hostname of the machine where the Python interpreter is currently executing.

If you want to know the current machine's IP address, you may want to use gethostbyname(gethostname()). This operation assumes that there is a valid address-to-host mapping for the host, and the assumption does not always hold.

Note: gethostname() doesn't always return the fully qualified domain name; use getfqdn() (see above).

socket.gethostbyaddr(ip_address)
Return a triple (hostname, aliaslist, ipaddrlist) where hostname is the primary host name responding to the given ip_address, aliaslist is a (possibly empty) list of alternative host names for the same address, and ipaddrlist is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function getfqdn(). gethostbyaddr() supports both IPv4 and IPv6.

socket.getnameinfo(sockaddr, flags)
Translate a socket address sockaddr into a 2-tuple (host, port). Depending on the settings of flags, the result can contain a fully-qualified domain name or numeric address representation in host. Similarly, port can contain a string port name or a numeric port number.

New in version 2.2.

socket.getprotobyname(protocolname)
Translate an Internet protocol name (for example, 'icmp') to a constant suitable for passing as the (optional) third argument to the socket() function. This is usually only needed for sockets opened in "raw" mode (SOCK_RAW); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

socket.getservbyname(servicename[, protocolname])
Translate an Internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be 'tcp' or 'udp', otherwise any protocol will match.

socket.getservbyport(port[, protocolname])
Translate an Internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be 'tcp' or 'udp', otherwise any protocol will match.

socket.socket([family[, type[, proto]]])
Create a new socket using the given address family, socket type and protocol number. The address family should be AF_INET (the default), AF_INET6 or AF_UNIX. The socket type should be SOCK_STREAM (the default), SOCK_DGRAM or perhaps one of the other SOCK_ constants. The protocol number is usually zero and may be omitted in that case.

socket.socketpair([family[, type[, proto]]])
Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the socket() function above. The default family is AF_UNIX if defined on the platform; otherwise, the default is AF_INET. Availability: Unix.

New in version 2.4.

socket.fromfd(fd, family, type[, proto])
Duplicate the file descriptor fd (an integer as returned by a file object's fileno() method) and build a socket object from the result. Address family, socket type and protocol number are as for the socket() function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode. Availability: Unix.

socket.ntohl(x)
Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.ntohs(x)
Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

socket.htonl(x)
Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.htons(x)
Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

socket.inet_aton(ip_string)
Convert an IPv4 address from dotted-quad string format (for example, '123.45.67.89') to 32-bit packed binary format, as a string four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type struct in_addr, which is the C type for the 32-bit packed binary this function returns.

inet_aton() also accepts strings with less than three dots; see the Unix manual page inet(3) for details.

If the IPv4 address string passed to this function is invalid, socket.error will be raised. Note that exactly what is valid depends on the underlying C implementation of inet_aton().

inet_aton() does not support IPv6, and inet_pton() should be used instead for IPv4/v6 dual stack support.

socket.inet_ntoa(packed_ip)
Convert a 32-bit packed IPv4 address (a string four characters in length) to its standard dotted-quad string representation (for example, '123.45.67.89'). This is useful when conversing with a program that uses the standard C library and needs objects of type struct in_addr, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the string passed to this function is not exactly 4 bytes in length, socket.error will be raised. inet_ntoa() does not support IPv6, and inet_ntop() should be used instead for IPv4/v6 dual stack support.

socket.inet_pton(address_family, ip_string)
Convert an IP address from its family-specific string format to a packed, binary format. inet_pton()

is useful when a library or network protocol calls for an object of type struct in_addr (similar to inet_aton()) or struct in6_addr.

Supported values for address_family are currently AF_INET and AF_INET6. If the IP address string ip_string is invalid, socket.error will be raised. Note that exactly what is valid depends on both the value of address_family and the underlying implementation of inet_pton().

Availability: Unix (maybe not all platforms).

New in version 2.3.

socket.inet_ntop(address_family, packed_ip)
Convert a packed IP address (a string of some number of characters) to its standard, family-specific string representation (for example, '7.10.0.5' or '5aef:2b::8') inet_ntop() is useful when a library or network protocol returns an object of type struct in_addr (similar to inet_ntoa()) or struct in6_addr.

Supported values for address_family are currently AF_INET and AF_INET6. If the string packed_ip is not the correct length for the specified address family, ValueError will be raised. A socket.error is raised for errors from the call to inet_ntop().

Availability: Unix (maybe not all platforms).

New in version 2.3.

socket.getdefaulttimeout()
Return the default timeout in seconds (float) for new socket objects. A value of None indicates that new socket objects have no timeout. When the socket module is first imported, the default is None.

New in version 2.3.

socket.setdefaulttimeout(timeout)
Set the default timeout in seconds (float) for new socket objects. A value of None indicates that new socket objects have no timeout. When the socket module is first imported, the default is None.

New in version 2.3.

socket.SocketType
This is a Python type object that represents the socket object type. It is the same as type(socket(...)).

See also:

Module SocketServer Classes that simplify writing network servers.

Module ssl A TLS/SSL wrapper for socket objects.

17.2.1 Socket Objects

Socket objects have the following methods. Except for makefile() these correspond to Unix system calls applicable to sockets.

socket.accept()
Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

socket.bind(address)
Bind the socket to address. The socket must not already be bound. (The format of address depends on the address family — see above.)

Note: This method has historically accepted a pair of parameters for AF_INET addresses instead of only a tuple. This was never intentional and is no longer available in Python 2.0 and later.

socket.close()
Close the socket. All future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

Note: close() releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call shutdown() before close().

socket.connect(address)
Connect to a remote socket at address. (The format of address depends on the address family — see above.)

Note: This method has historically accepted a pair of parameters for AF_INET addresses instead of only a tuple. This was never intentional and is no longer available in Python 2.0 and later.

socket.connect_ex(address)
Like connect(address), but return an error indicator instead of raising an exception for errors returned by the C-level connect() call (other problems, such as "host not found," can still raise exceptions). The error indicator is 0 if the operation succeeded, otherwise the value of the errno variable. This is useful to support, for example, asynchronous connects.

Note: This method has historically accepted a pair of parameters for AF_INET addresses instead of only a tuple. This was never intentional and is no longer available in Python 2.0 and later.

socket.fileno()
Return the socket's file descriptor (a small integer). This is useful with select.select().

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as os.fdopen()). Unix does not have this limitation.

socket.getpeername()
Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

socket.getsockname()
Return the socket's own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.)

socket.getsockopt(level, optname[, buflen])
Return the value of the given socket option (see the Unix man page getsockopt(2)). The needed symbolic constants (SO_* etc.) are defined in this module. If buflen is absent, an integer option is assumed and its integer value is returned by the function. If buflen is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a string. It is up to the caller to decode the contents of the buffer (see the optional built-in module struct for a way to decode C structures encoded as strings).

socket.ioctl(control, option)

Platform Windows

The ioctl() method is a limited interface to the WSAIoctl system interface. Please refer to the Win32 documentation for more information.

On other platforms, the generic fcntl.fcntl() and fcntl.ioctl() functions may be used; they accept a socket object as their first argument.

New in version 2.6.

socket.listen(backlog)
Listen for connections made to the socket. The backlog argument specifies the maximum number of queued connections and should be at least 0; the maximum value is system-dependent (usually 5), the minimum value is forced to 0.

socket.makefile([mode[, bufsize]])
Return a file object associated with the socket. (File objects are described in File Objects.) The file object does not close the socket explicitly when its close() method is called, but only removes its reference to the socket object, so that the socket will be closed if it is not referenced from anywhere else.

The socket must be in blocking mode (it can not have a timeout). The optional mode and bufsize arguments are interpreted the same way as by the built-in file() function.

Note: On Windows, the file-like object created by makefile() cannot be used where a file object with a file descriptor is expected, such as the stream arguments of subprocess.Popen().

---

socket.recv(bufsize[, flags])
Receive data from the socket. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by bufsize. See the Unix manual page recv(2) for the meaning of the optional argument flags; it defaults to zero.

Note: For best match with hardware and network realities, the value of bufsize should be a relatively small power of 2, for example, 4096.

---

socket.recvfrom(bufsize[, flags])
Receive data from the socket. The return value is a pair (string, address) where string is a string representing the data received and address is the address of the socket sending the data. See the Unix manual page recv(2) for the meaning of the optional argument flags; it defaults to zero. (The format of address depends on the address family — see above.)

socket.recvfrom_into(buffer[, nbytes[, flags]])
Receive data from the socket, writing it into buffer instead of creating a new string. The return value is a pair (nbytes, address) where nbytes is the number of bytes received and address is the address of the socket sending the data. See the Unix manual page recv(2) for the meaning of the optional argument flags; it defaults to zero. (The format of address depends on the address family — see above.)

New in version 2.5.

socket.recv_into(buffer[, nbytes[, flags]])
Receive up to nbytes bytes from the socket, storing the data into a buffer rather than creating a new string. If nbytes is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page recv(2) for the meaning of the optional argument flags; it defaults to zero.

New in version 2.5.

socket.send(string[, flags])
: Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for recv() above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this concept, consult the socket-howto.

socket.sendall(string[, flags])
: Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for recv() above. Unlike send(), this method continues to send data from string until either all data has been sent or an error occurs. None is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

socket.sendto(string, address)
socket.sendto(string, flags, address)
: Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by address. The optional flags argument has the same meaning as for recv() above. Return the number of bytes sent. (The format of address depends on the address family — see above.)

socket.setblocking(flag)
: Set blocking or non-blocking mode of the socket: if flag is 0, the socket is set to non-blocking, else to blocking mode. Initially all sockets are in blocking mode. In non-blocking mode, if a recv() call doesn't find any data, or if a send() call can't immediately dispose of the data, an error exception is raised; in blocking mode, the calls block until they can proceed. s.setblocking(0) is equivalent to s.settimeout(0.0); s.setblocking(1) is equivalent to s.settimeout(None).

socket.settimeout(value)
: Set a timeout on blocking socket operations. The value argument can be a nonnegative float expressing seconds, or None. If a float is given, subsequent socket operations will raise a timeout exception if the timeout period value has elapsed before the operation has completed. Setting a timeout of None disables timeouts on socket operations. s.settimeout(0.0) is equivalent to s.setblocking(0); s.settimeout(None) is equivalent to s.setblocking(1).

  New in version 2.3.

socket.gettimeout()
: Return the timeout in seconds (float) associated with socket operations, or None if no timeout is set. This reflects the last call to setblocking() or settimeout().

  New in version 2.3.

Some notes on socket blocking and timeouts: A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are always created in blocking mode. In blocking mode, operations block until complete or the system returns an error (such as connection timed out). In non-blocking mode, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately. In timeout mode, operations fail if they cannot be completed within the timeout specified for the socket or if the system returns an error. The setblocking() method is simply a shorthand for certain settimeout() calls.

Timeout mode internally sets the socket in non-blocking mode. The blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. A consequence of this is that file objects returned by the makefile() method must only be used when the socket is in blocking mode; in timeout or non-blocking mode file operations that cannot be completed immediately will fail.

Note that the connect() operation is subject to the timeout setting, and in general it is recommended to call settimeout() before calling connect() or pass a timeout parameter to create_connection(). The system network stack may return a connection timeout error of its own regardless of any Python socket timeout setting.

socket.setsockopt(level, optname, value)

Set the value of the given socket option (see the Unix manual page setsockopt(2)). The needed symbolic constants are defined in the socket module (SO_* etc.). The value can be an integer or a string representing a buffer. In the latter case it is up to the caller to ensure that the string contains the proper bits (see the optional built-in module struct for a way to encode C structures as strings).

socket.shutdown(how)
Shut down one or both halves of the connection. If how is SHUT_RD, further receives are disallowed. If how is SHUT_WR, further sends are disallowed. If how is SHUT_RDWR, further sends and receives are disallowed. Depending on the platform, shutting down one half of the connection can also close the opposite half (e.g. on Mac OS X, shutdown(SHUT_WR) does not allow further reads on the other end of the connection).

Note that there are no methods read() or write(); use recv() and send() without flags argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the socket constructor.

socket.family
The socket family.

New in version 2.5.

socket.type
The socket type.

New in version 2.5.

socket.proto
The socket protocol.

New in version 2.5.

17.2.2 Example

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence socket(), bind(), listen(), accept() (possibly repeating the accept() to service more than one client), while a client only needs the sequence socket(), connect(). Also note that the server does not sendall()/recv() on the socket it is listening on but on the new socket returned by accept().

The first two examples support IPv4 only.

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
conn.close()
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.sendall('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to the all addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except socket.error as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print 'could not open socket'
    sys.exit(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```

```
# Echo client program
import socket
import sys

```

(continues on next page)

(continued from previous page)

```
HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except socket.error as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print 'could not open socket'
    sys.exit(1)
s.sendall('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

The last example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface:

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print s.recvfrom(65565)

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

Running an example several times with too small delay between executions, could lead to this error:

```
socket.error: [Errno 98] Address already in use
```

This is because the previous execution has left the socket in a TIME_WAIT state, and can't be immediately reused.

There is a socket flag to set, in order to prevent this, socket.SO_REUSEADDR:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

the SO_REUSEADDR flag tells the kernel to reuse a local socket in TIME_WAIT state, without waiting for its natural timeout to expire.

17.3 ssl — TLS/SSL wrapper for socket objects

New in version 2.6.

Source code: Lib/ssl.py

This module provides access to Transport Layer Security (often known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library. It is available on all modern Unix systems, Windows, Mac OS X, and probably additional platforms, as long as OpenSSL is installed on that platform.

Changed in version 2.7.13: Updated to support linking with OpenSSL 1.1.0

Note: Some behavior may be platform dependent, since calls are made to the operating system socket APIs. The installed version of OpenSSL may also cause variations in behavior. For example, TLSv1.1 and TLSv1.2 come with openssl version 1.0.1.

Warning: Don't use this module without reading the Security considerations. Doing so may lead to a false sense of security, as the default settings of the ssl module are not necessarily appropriate for your application.

This section documents the objects and functions in the ssl module; for more general information about TLS, SSL, and certificates, the reader is referred to the documents in the "See Also" section at the bottom.

This module provides a class, ssl.SSLSocket, which is derived from the socket.socket type, and provides a socket-like wrapper that also encrypts and decrypts the data going over the socket with SSL. It supports additional methods such as getpeercert(), which retrieves the certificate of the other side of the connection, and cipher(),which retrieves the cipher being used for the secure connection.

For more sophisticated applications, the ssl.SSLContext class helps manage settings and certificates, which can then be inherited by SSL sockets created through the SSLContext.wrap_socket() method.

17.3.1 Functions, Constants, and Exceptions

exception ssl.SSLError

Raised to signal an error from the underlying SSL implementation (currently provided by the OpenSSL library). This signifies some problem in the higher-level encryption and authentication layer that's superimposed on the underlying network connection. This error is a subtype of socket.error, which in turn is a subtype of IOError. The error code and message of SSLError instances are provided by the OpenSSL library.

library
: A string mnemonic designating the OpenSSL submodule in which the error occurred, such as SSL, PEM or X509. The range of possible values depends on the OpenSSL version.

  New in version 2.7.9.

reason
: A string mnemonic designating the reason this error occurred, for example CERTIFICATE_VERIFY_FAILED. The range of possible values depends on the OpenSSL version.

  New in version 2.7.9.

exception ssl.SSLZeroReturnError
: A subclass of SSLError raised when trying to read or write and the SSL connection has been closed cleanly. Note that this doesn't mean that the underlying transport (read TCP) has been closed.

  New in version 2.7.9.

exception ssl.SSLWantReadError
: A subclass of SSLError raised by a non-blocking SSL socket when trying to read or write data, but more data needs to be received on the underlying TCP transport before the request can be fulfilled.

  New in version 2.7.9.

exception ssl.SSLWantWriteError
: A subclass of SSLError raised by a non-blocking SSL socket when trying to read or write data, but more data needs to be sent on the underlying TCP transport before the request can be fulfilled.

  New in version 2.7.9.

exception ssl.SSLSyscallError
: A subclass of SSLError raised when a system error was encountered while trying to fulfill an operation on a SSL socket. Unfortunately, there is no easy way to inspect the original errno number.

  New in version 2.7.9.

exception ssl.SSLEOFError
: A subclass of SSLError raised when the SSL connection has been terminated abruptly. Generally, you shouldn't try to reuse the underlying transport when this error is encountered.

  New in version 2.7.9.

exception ssl.CertificateError
: Raised to signal an error with a certificate (such as mismatching hostname). Certificate errors detected by OpenSSL, though, raise an SSLError.

Socket creation

The following function allows for standalone socket creation. Starting from Python 2.7.9, it can be more flexible to use SSLContext.wrap_socket() instead.

ssl.wrap_socket(sock, keyfile=None, certfile=None, server_side=False, cert_reqs=CERT_NONE, ssl_version={see docs}, ca_certs=None, do_handshake_on_connect=True, suppress_ragged_eofs=True, ciphers=None)
: Takes an instance sock of socket.socket, and returns an instance of ssl.SSLSocket, a subtype of socket.socket, which wraps the underlying socket in an SSL context. sock must be a SOCK_STREAM socket; other socket types are unsupported.

  For client-side sockets, the context construction is lazy; if the underlying socket isn't connected yet, the context construction will be performed after connect() is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping

is automatically performed on client connections accepted via the accept() method. wrap_socket() may raise SSLError.

The keyfile and certfile parameters specify optional files which contain a certificate to be used to identify the local side of the connection. See the discussion of Certificates for more information on how the certificate is stored in the certfile.

The parameter server_side is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

The parameter cert_reqs specifies whether a certificate is required from the other side of the connection, and whether it will be validated if provided. It must be one of the three values CERT_NONE (certificates ignored), CERT_OPTIONAL (not required, but validated if provided), or CERT_REQUIRED (required and validated). If the value of this parameter is not CERT_NONE, then the ca_certs parameter must point to a file of CA certificates.

The ca_certs file contains a set of concatenated "certification authority" certificates, which are used to validate certificates passed from the other end of the connection. See the discussion of Certificates for more information about how to arrange the certificates in this file.

The parameter ssl_version specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server's choice. Most of the versions are not interoperable with the other versions. If not specified, the default is PROTOCOL_SSLv23; it provides the most compatibility with other versions.

Here's a table showing which versions in a client (down the side) can connect to which versions in a server (along the top):

client / server	SSLv2	SSLv3	SSLv23	TLSv1	TLSv1.1	TLSv1.2
SSLv2	yes	no	yes	no	no	no
SSLv3	no	yes	yes	no	no	no
SSLv23[1]	no	yes	yes	yes	yes	yes
TLSv1	no	no	yes	yes	no	no
TLSv1.1	no	no	yes	no	yes	no
TLSv1.2	no	no	yes	no	no	yes

Note: Which connections succeed will vary depending on the version of OpenSSL. For example, before OpenSSL 1.0.0, an SSLv23 client would always attempt SSLv2 connections.

The ciphers parameter sets the available ciphers for this SSL object. It should be a string in the OpenSSL cipher list format.

The parameter do_handshake_on_connect specifies whether to do the SSL handshake automatically after doing a socket.connect(), or whether the application program will call it explicitly, by invoking the SSLSocket.do_handshake() method. Calling SSLSocket.do_handshake() explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter suppress_ragged_eofs specifies how the SSLSocket.read() method should signal unexpected EOF from the other end of the connection. If specified as True (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying socket; if False, it will raise the exceptions back to the caller.

Changed in version 2.7: New optional argument ciphers.

[1] TLS 1.3 protocol will be available with PROTOCOL_SSLv23 in OpenSSL >= 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.

Context creation

A convenience function helps create SSLContext objects for common purposes.

ssl.create_default_context(purpose=Purpose.SERVER_AUTH, cafile=None, capath=None, cadata=None)

Return a new SSLContext object with default settings for the given purpose. The settings are chosen by the ssl module, and usually represent a higher security level than when calling the SSLContext constructor directly.

cafile, capath, cadata represent optional CA certificates to trust for certificate verification, as in SSLContext.load_verify_locations(). If all three are None, this function can choose to trust the system's default CA certificates instead.

The settings are: PROTOCOL_SSLv23, OP_NO_SSLv2, and OP_NO_SSLv3 with high encryption cipher suites without RC4 and without unauthenticated cipher suites. Passing SERVER_AUTH as purpose sets verify_mode to CERT_REQUIRED and either loads CA certificates (when at least one of cafile, capath or cadata is given) or uses SSLContext.load_default_certs() to load default CA certificates.

Note: The protocol, options, cipher and other settings may change to more restrictive values anytime without prior deprecation. The values represent a fair balance between compatibility and security.

If your application needs specific settings, you should create a SSLContext and apply the settings yourself.

Note: If you find that when certain older clients or servers attempt to connect with a SSLContext created by this function that they get an error stating "Protocol or cipher suite mismatch", it may be that they only support SSL3.0 which this function excludes using the OP_NO_SSLv3. SSL3.0 is widely considered to be completely broken. If you still wish to continue to use this function but still allow SSL 3.0 connections you can re-enable them using:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

New in version 2.7.9.

Changed in version 2.7.10: RC4 was dropped from the default cipher string.

Changed in version 2.7.13: ChaCha20/Poly1305 was added to the default cipher string.

3DES was dropped from the default cipher string.

ssl._https_verify_certificates(enable=True)

Specifies whether or not server certificates are verified when creating client HTTPS connections without specifying a particular SSL context.

Starting with Python 2.7.9, httplib and modules which use it, such as urllib2 and xmlrpclib, default to verifying remote server certificates received when establishing client HTTPS connections. This default verification checks that the certificate is signed by a Certificate Authority in the system trust store and that the Common Name (or Subject Alternate Name) on the presented certificate matches the requested host.

Setting enable to True ensures this default behaviour is in effect.

Setting enable to False reverts the default HTTPS certificate handling to that of Python 2.7.8 and earlier, allowing connections to servers using self-signed certificates, servers using certificates signed by

a Certicate Authority not present in the system trust store, and servers where the hostname does not match the presented server certificate.

The leading underscore on this function denotes that it intentionally does not exist in any implementation of Python 3 and may not be present in all Python 2.7 implementations. The portable approach to bypassing certificate checks or the system trust store when necessary is for tools to enable that on a case-by-case basis by explicitly passing in a suitably configured SSL context, rather than reverting the default behaviour of the standard library client modules.

New in version 2.7.12.

See also:

- CVE-2014-9365 – HTTPS man-in-the-middle attack against Python clients using default settings
- PEP 476 – Enabling certificate verification by default for HTTPS
- PEP 493 – HTTPS verification migration tools for Python 2.7

Random generation

Deprecated since version 2.7.13: OpenSSL has deprecated ssl.RAND_pseudo_bytes(), use ssl.RAND_bytes() instead.

ssl.RAND_status()
Return True if the SSL pseudo-random number generator has been seeded with 'enough' randomness, and False otherwise. You can use ssl.RAND_egd() and ssl.RAND_add() to increase the randomness of the pseudo-random number generator.

ssl.RAND_egd(path)
If you are running an entropy-gathering daemon (EGD) somewhere, and path is the pathname of a socket connection open to it, this will read 256 bytes of randomness from the socket, and add it to the SSL pseudo-random number generator to increase the security of generated secret keys. This is typically only necessary on systems without better sources of randomness.

See http://egd.sourceforge.net/ or http://prngd.sourceforge.net/ for sources of entropy-gathering daemons.

Availability: not available with LibreSSL and OpenSSL > 1.1.0

ssl.RAND_add(bytes, entropy)
Mix the given bytes into the SSL pseudo-random number generator. The parameter entropy (a float) is a lower bound on the entropy contained in string (so you can always use 0.0). See RFC 1750 for more information on sources of entropy.

Certificate handling

ssl.match_hostname(cert, hostname)
Verify that cert (in decoded format as returned by SSLSocket.getpeercert()) matches the given hostname. The rules applied are those for checking the identity of HTTPS servers as outlined in RFC 2818 and RFC 6125, except that IP addresses are not currently supported. In addition to HTTPS, this function should be suitable for checking the identity of servers in various SSL-based protocols such as FTPS, IMAPS, POPS and others.

CertificateError is raised on failure. On success, the function returns nothing:

```
>>> cert = {'subject': ((('commonName', 'example.com'),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

New in version 2.7.9.

ssl.cert_time_to_seconds(cert_time)
Return the time in seconds since the Epoch, given the cert_time string representing the "notBefore" or "notAfter" date from a certificate in "%b %d %H:%M:%S %Y %Z" strptime format (C locale).

Here's an example:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

"notBefore" or "notAfter" dates must use GMT (RFC 5280).

Changed in version 2.7.9: Interpret the input time as a time in UTC as specified by 'GMT' timezone in the input string. Local timezone was used previously. Return an integer (no fractions of a second in the input format)

ssl.get_server_certificate(addr, ssl_version=PROTOCOL_SSLv23, ca_certs=None)
Given the address addr of an SSL-protected server, as a (hostname, port-number) pair, fetches the server's certificate, and returns it as a PEM-encoded string. If ssl_version is specified, uses that version of the SSL protocol to attempt to connect to the server. If ca_certs is specified, it should be a file containing a list of root certificates, the same format as used for the same parameter in wrap_socket(). The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails.

Changed in version 2.7.9: This function is now IPv6-compatible, and the default ssl_version is changed from PROTOCOL_SSLv3 to PROTOCOL_SSLv23 for maximum compatibility with modern servers.

ssl.DER_cert_to_PEM_cert(DER_cert_bytes)
Given a certificate as a DER-encoded blob of bytes, returns a PEM-encoded string version of the same certificate.

ssl.PEM_cert_to_DER_cert(PEM_cert_string)
Given a certificate as an ASCII PEM string, returns a DER-encoded sequence of bytes for that same certificate.

ssl.get_default_verify_paths()
Returns a named tuple with paths to OpenSSL's default cafile and capath. The paths are the same as used by SSLContext.set_default_verify_paths(). The return value is a named tuple DefaultVerifyPaths:

- cafile - resolved path to cafile or None if the file doesn't exist,
- capath - resolved path to capath or None if the directory doesn't exist,
- openssl_cafile_env - OpenSSL's environment key that points to a cafile,
- openssl_cafile - hard coded path to a cafile,

- openssl_capath_env - OpenSSL's environment key that points to a capath,
- openssl_capath - hard coded path to a capath directory

Availability: LibreSSL ignores the environment vars openssl_cafile_env and openssl_capath_env

New in version 2.7.9.

ssl.enum_certificates(store_name)

Retrieve certificates from Windows' system cert store. store_name may be one of CA, ROOT or MY. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either x509_asn for X.509 ASN.1 data or pkcs_7_asn for PKCS#7 ASN.1 data. Trust specifies the purpose of the certificate as a set of OIDS or exactly True if the certificate is trustworthy for all purposes.

Example:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

Availability: Windows.

New in version 2.7.9.

ssl.enum_crls(store_name)

Retrieve CRLs from Windows' system cert store. store_name may be one of CA, ROOT or MY. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either x509_asn for X.509 ASN.1 data or pkcs_7_asn for PKCS#7 ASN.1 data.

Availability: Windows.

New in version 2.7.9.

Constants

ssl.CERT_NONE

Possible value for SSLContext.verify_mode, or the cert_reqs parameter to wrap_socket(). In this mode (the default), no certificates will be required from the other side of the socket connection. If a certificate is received from the other end, no attempt to validate it is made.

See the discussion of Security considerations below.

ssl.CERT_OPTIONAL

Possible value for SSLContext.verify_mode, or the cert_reqs parameter to wrap_socket(). In this mode no certificates will be required from the other side of the socket connection; but if they are provided, validation will be attempted and an SSLError will be raised on failure.

Use of this setting requires a valid set of CA certificates to be passed, either to SSLContext.load_verify_locations() or as a value of the ca_certs parameter to wrap_socket().

ssl.CERT_REQUIRED

Possible value for SSLContext.verify_mode, or the cert_reqs parameter to wrap_socket(). In this mode, certificates are required from the other side of the socket connection; an SSLError will be raised if no certificate is provided, or if its validation fails.

Use of this setting requires a valid set of CA certificates to be passed, either to SSLContext.load_verify_locations() or as a value of the ca_certs parameter to wrap_socket().

ssl.VERIFY_DEFAULT
Possible value for SSLContext.verify_flags. In this mode, certificate revocation lists (CRLs) are not checked. By default OpenSSL does neither require nor verify CRLs.

New in version 2.7.9.

ssl.VERIFY_CRL_CHECK_LEAF
Possible value for SSLContext.verify_flags. In this mode, only the peer cert is check but non of the intermediate CA certificates. The mode requires a valid CRL that is signed by the peer cert's issuer (its direct ancestor CA). If no proper has been loaded SSLContext.load_verify_locations, validation will fail.

New in version 2.7.9.

ssl.VERIFY_CRL_CHECK_CHAIN
Possible value for SSLContext.verify_flags. In this mode, CRLs of all certificates in the peer cert chain are checked.

New in version 2.7.9.

ssl.VERIFY_X509_STRICT
Possible value for SSLContext.verify_flags to disable workarounds for broken X.509 certificates.

New in version 2.7.9.

ssl.VERIFY_X509_TRUSTED_FIRST
Possible value for SSLContext.verify_flags. It instructs OpenSSL to prefer trusted certificates when building the trust chain to validate a certificate. This flag is enabled by default.

New in version 2.7.10.

ssl.PROTOCOL_TLS
Selects the highest protocol version that both the client and server support. Despite the name, this option can select "TLS" protocols as well as "SSL".

New in version 2.7.13.

ssl.PROTOCOL_SSLv23
Alias for PROTOCOL_TLS.

Deprecated since version 2.7.13: Use PROTOCOL_TLS instead.

ssl.PROTOCOL_SSLv2
Selects SSL version 2 as the channel encryption protocol.

This protocol is not available if OpenSSL is compiled with the OPENSSL_NO_SSL2 flag.

> Warning: SSL version 2 is insecure. Its use is highly discouraged.

Deprecated since version 2.7.13: OpenSSL has removed support for SSLv2.

ssl.PROTOCOL_SSLv3
Selects SSL version 3 as the channel encryption protocol.

This protocol is not be available if OpenSSL is compiled with the OPENSSL_NO_SSLv3 flag.

Warning: SSL version 3 is insecure. Its use is highly discouraged.

Deprecated since version 2.7.13: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like OP_NO_SSLv3 instead.

ssl.PROTOCOL_TLSv1
Selects TLS version 1.0 as the channel encryption protocol.

Deprecated since version 2.7.13: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like OP_NO_SSLv3 instead.

ssl.PROTOCOL_TLSv1_1
Selects TLS version 1.1 as the channel encryption protocol. Available only with openssl version 1.0.1+.

New in version 2.7.9.

Deprecated since version 2.7.13: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like OP_NO_SSLv3 instead.

ssl.PROTOCOL_TLSv1_2
Selects TLS version 1.2 as the channel encryption protocol. This is the most modern version, and probably the best choice for maximum protection, if both sides can speak it. Available only with openssl version 1.0.1+.

New in version 2.7.9.

Deprecated since version 2.7.13: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like OP_NO_SSLv3 instead.

ssl.OP_ALL
Enables workarounds for various bugs present in other SSL implementations. This option is set by default. It does not necessarily set the same flags as OpenSSL's SSL_OP_ALL constant.

New in version 2.7.9.

ssl.OP_NO_SSLv2
Prevents an SSLv2 connection. This option is only applicable in conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing SSLv2 as the protocol version.

New in version 2.7.9.

ssl.OP_NO_SSLv3
Prevents an SSLv3 connection. This option is only applicable in conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing SSLv3 as the protocol version.

New in version 2.7.9.

ssl.OP_NO_TLSv1
Prevents a TLSv1 connection. This option is only applicable in conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing TLSv1 as the protocol version.

New in version 2.7.9.

ssl.OP_NO_TLSv1_1
Prevents a TLSv1.1 connection. This option is only applicable in conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing TLSv1.1 as the protocol version. Available only with openssl version 1.0.1+.

New in version 2.7.9.

ssl.OP_NO_TLSv1_2
Prevents a TLSv1.2 connection. This option is only applicable in conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing TLSv1.2 as the protocol version. Available only with openssl version 1.0.1+.

New in version 2.7.9.

ssl.OP_NO_TLSv1_3
Prevents a TLSv1.3 connection. This option is only applicable in conjunction with PROTOCOL_TLS. It prevents the peers from choosing TLSv1.3 as the protocol version. TLS 1.3 is available with OpenSSL 1.1.1 or later. When Python has been compiled against an older version of OpenSSL, the flag defaults to 0.

New in version 2.7.15.

ssl.OP_CIPHER_SERVER_PREFERENCE
Use the server's cipher ordering preference, rather than the client's. This option has no effect on client sockets and SSLv2 server sockets.

New in version 2.7.9.

ssl.OP_SINGLE_DH_USE
Prevents re-use of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

New in version 2.7.9.

ssl.OP_SINGLE_ECDH_USE
Prevents re-use of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

New in version 2.7.9.

ssl.OP_ENABLE_MIDDLEBOX_COMPAT
Send dummy Change Cipher Spec (CCS) messages in TLS 1.3 handshake to make a TLS 1.3 connection look more like a TLS 1.2 connection.

This option is only available with OpenSSL 1.1.1 and later.

New in version 2.7.16.

ssl.OP_NO_COMPRESSION
Disable compression on the SSL channel. This is useful if the application protocol supports its own compression scheme.

This option is only available with OpenSSL 1.0.0 and later.

New in version 2.7.9.

ssl.HAS_ALPN
Whether the OpenSSL library has built-in support for the Application-Layer Protocol Negotiation TLS extension as described in RFC 7301.

New in version 2.7.10.

ssl.HAS_ECDH
Whether the OpenSSL library has built-in support for Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.

New in version 2.7.9.

ssl.HAS_SNI
Whether the OpenSSL library has built-in support for the Server Name Indication extension (as defined in RFC 4366).

New in version 2.7.9.

ssl.HAS_NPN
: Whether the OpenSSL library has built-in support for Next Protocol Negotiation as described in the NPN draft specification. When true, you can use the SSLContext.set_npn_protocols() method to advertise which protocols you want to support.

  New in version 2.7.9.

ssl.HAS_TLSv1_3
: Whether the OpenSSL library has built-in support for the TLS 1.3 protocol.

  New in version 2.7.15.

ssl.CHANNEL_BINDING_TYPES
: List of supported TLS channel binding types. Strings in this list can be used as arguments to SSLSocket.get_channel_binding().

  New in version 2.7.9.

ssl.OPENSSL_VERSION
: The version string of the OpenSSL library loaded by the interpreter:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 0.9.8k 25 Mar 2009'
```

  New in version 2.7.

ssl.OPENSSL_VERSION_INFO
: A tuple of five integers representing version information about the OpenSSL library:

```
>>> ssl.OPENSSL_VERSION_INFO
(0, 9, 8, 11, 15)
```

  New in version 2.7.

ssl.OPENSSL_VERSION_NUMBER
: The raw version number of the OpenSSL library, as a single integer:

```
>>> ssl.OPENSSL_VERSION_NUMBER
9470143L
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x9080bfL'
```

  New in version 2.7.

ssl.ALERT_DESCRIPTION_HANDSHAKE_FAILURE
ssl.ALERT_DESCRIPTION_INTERNAL_ERROR
ALERT_DESCRIPTION_*
: Alert Descriptions from RFC 5246 and others. The IANA TLS Alert Registry contains this list and references to the RFCs where their meaning is defined.

  Used as the return value of the callback function in SSLContext.set_servername_callback().

  New in version 2.7.9.

Purpose.SERVER_AUTH
: Option for create_default_context() and SSLContext.load_default_certs(). This value indicates that the context may be used to authenticate Web servers (therefore, it will be used to create client-side sockets).

  New in version 2.7.9.

Purpose.CLIENT_AUTH
: Option for create_default_context() and SSLContext.load_default_certs(). This value indicates that the context may be used to authenticate Web clients (therefore, it will be used to create server-side sockets).

  New in version 2.7.9.

17.3.2 SSL Sockets

SSL sockets provide the following methods of Socket Objects:

- accept()
- bind()
- close()
- connect()
- fileno()
- getpeername(), getsockname()
- getsockopt(), setsockopt()
- gettimeout(), settimeout(), setblocking()
- listen()
- makefile()
- recv(), recv_into() (but passing a non-zero flags argument is not allowed)
- send(), sendall() (with the same limitation)
- shutdown()

However, since the SSL (and TLS) protocol has its own framing atop of TCP, the SSL sockets abstraction can, in certain respects, diverge from the specification of normal, OS-level sockets. See especially the notes on non-blocking sockets.

SSL sockets also have the following additional methods and attributes:

SSLSocket.do_handshake()
: Perform the SSL setup handshake.

  Changed in version 2.7.9: The handshake method also performs match_hostname() when the check_hostname attribute of the socket's context is true.

SSLSocket.getpeercert(binary_form=False)
: If there is no certificate for the peer on the other end of the connection, return None. If the SSL handshake hasn't been done yet, raise ValueError.

  If the binary_form parameter is False, and a certificate was received from the peer, this method returns a dict instance. If the certificate was not validated, the dict is empty. If the certificate was validated, it returns a dict with several keys, amongst them subject (the principal for which the certificate was issued) and issuer (the principal issuing the certificate). If a certificate contains an instance of the Subject Alternative Name extension (see RFC 3280), there will also be a subjectAltName key in the dictionary.

  The subject and issuer fields are tuples containing the sequence of relative distinguished names (RDNs) given in the certificate's data structure for the respective fields, and each RDN is a sequence of name-value pairs. Here is a real-world example:

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

Note: To validate a certificate for a particular service, you can use the match_hostname() function.

If the binary_form parameter is True, and a certificate was provided, this method returns the DER-encoded form of the entire certificate as a sequence of bytes, or None if the peer did not provide a certificate. Whether the peer provides a certificate depends on the SSL socket's role:

- for a client SSL socket, the server will always provide a certificate, regardless of whether validation was required;
- for a server SSL socket, the client will only provide a certificate when requested by the server; therefore getpeercert() will return None if you used CERT_NONE (rather than CERT_OPTIONAL or CERT_REQUIRED).

Changed in version 2.7.9: The returned dictionary includes additional items such as issuer and notBefore. Additionall ValueError is raised when the handshake isn't done. The returned dictionary includes additional X509v3 extension items such as crlDistributionPoints, caIssuers and OCSP URIs.

SSLSocket.cipher()
Returns a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used. If no connection has been established, returns None.

SSLSocket.compression()
Return the compression algorithm being used as a string, or None if the connection isn't compressed.

If the higher-level protocol supports its own compression mechanism, you can use OP_NO_COMPRESSION to disable SSL-level compression.

New in version 2.7.9.

SSLSocket.get_channel_binding(cb_type="tls-unique")
Get channel binding data for current connection, as a bytes object. Returns None if not connected or the handshake has not been completed.

The cb_type parameter allow selection of the desired channel binding type. Valid channel binding types are listed in the CHANNEL_BINDING_TYPES list. Currently only the 'tls-unique' channel binding, defined by RFC 5929, is supported. ValueError will be raised if an unsupported channel binding type is requested.

New in version 2.7.9.

SSLSocket.selected_alpn_protocol()
Return the protocol that was selected during the TLS handshake. If SSLContext.set_alpn_protocols() was not called, if the other party does not support ALPN, if this socket does not support any of the client's proposed protocols, or if the handshake has not happened yet, None is returned.

New in version 2.7.10.

SSLSocket.selected_npn_protocol()
Return the higher-level protocol that was selected during the TLS/SSL handshake. If SSLContext.set_npn_protocols() was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return None.

New in version 2.7.9.

SSLSocket.unwrap()
Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The returned socket should always be used for further communication with the other side of the connection, rather than the original socket.

SSLSocket.version()
Return the actual SSL protocol version negotiated by the connection as a string, or None is no secure connection is established. As of this writing, possible return values include "SSLv2", "SSLv3", "TLSv1", "TLSv1.1" and "TLSv1.2". Recent OpenSSL versions may define more return values.

New in version 2.7.9.

SSLSocket.context
The SSLContext object this SSL socket is tied to. If the SSL socket was created using the top-level wrap_socket() function (rather than SSLContext.wrap_socket()), this is a custom context object created for this SSL socket.

New in version 2.7.9.

17.3.3 SSL Contexts

New in version 2.7.9.

An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

class ssl.SSLContext(protocol)
Create a new SSL context. You must pass protocol which must be one of the PROTOCOL_* constants defined in this module. PROTOCOL_SSLv23 is currently recommended for maximum interoperability.

See also:

create_default_context() lets the ssl module choose security settings for a given purpose.

Changed in version 2.7.16: The context is created with secure default values. The options OP_NO_COMPRESSION, OP_CIPHER_SERVER_PREFERENCE, OP_SINGLE_DH_USE, OP_SINGLE_ECDH_USE, OP_NO_SSLv2 (except for PROTOCOL_SSLv2), and OP_NO_SSLv3 (except for PROTOCOL_SSLv3) are set by default. The initial cipher suite list contains only HIGH ciphers, no NULL ciphers and no MD5 ciphers (except for PROTOCOL_SSLv2).

SSLContext objects have the following methods and attributes:

SSLContext.cert_store_stats()
Get statistics about quantities of loaded X.509 certificates, count of X.509 certificates flagged as CA certificates and certificate revocation lists as dictionary.

Example for a context with one CA cert and one other cert:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

SSLContext.load_cert_chain(certfile, keyfile=None, password=None)
Load a private key and the corresponding certificate. The certfile string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. The keyfile string, if present, must point to a file containing the private key in. Otherwise the private key will be taken from certfile as well. See the discussion of Certificates for more information on how the certificate is stored in the certfile.

The password argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the password argument. It will be ignored if the private key is not encrypted and no password is needed.

If the password argument is not specified and a password is required, OpenSSL's built-in password prompting mechanism will be used to interactively prompt the user for a password.

An SSLError is raised if the private key doesn't match with the certificate.

SSLContext.load_default_certs(purpose=Purpose.SERVER_AUTH)
Load a set of default "certification authority" (CA) certificates from default locations. On Windows it loads CA certs from the CA and ROOT system stores. On other systems it calls SSLContext.set_default_verify_paths(). In the future the method may load CA certificates from other locations, too.

The purpose flag specifies what kind of CA certificates are loaded. The default settings Purpose.SERVER_AUTH loads certificates, that are flagged and trusted for TLS web server authentication (client side sockets). Purpose.CLIENT_AUTH loads CA certificates for client certificate verification on the server side.

SSLContext.load_verify_locations(cafile=None, capath=None, cadata=None)
Load a set of "certification authority" (CA) certificates used to validate other peers' certificates when verify_mode is other than CERT_NONE. At least one of cafile or capath must be specified.

This method can also load certification revocation lists (CRLs) in PEM or DER format. In order to make use of CRLs, SSLContext.verify_flags must be configured properly.

The cafile string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of Certificates for more information about how to arrange the certificates in this file.

The capath string, if present, is the path to a directory containing several CA certificates in PEM format, following an OpenSSL specific layout.

The cadata object, if present, is either an ASCII string of one or more PEM-encoded certificates or a bytes-like object of DER-encoded certificates. Like with capath extra lines around PEM-encoded certificates are ignored but at least one certificate must be present.

SSLContext.get_ca_certs(binary_form=False)
Get a list of loaded "certification authority" (CA) certificates. If the binary_form parameter is False each list entry is a dict like the output of SSLSocket.getpeercert(). Otherwise the method returns a

list of DER-encoded certificates. The returned list does not contain certificates from capath unless a certificate was requested and loaded by a SSL connection.

Note: Certificates in a capath directory aren't loaded unless they have been used at least once.

SSLContext.set_default_verify_paths()
Load a set of default "certification authority" (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there's no easy way to know whether this method succeeds: no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

SSLContext.set_ciphers(ciphers)
Set the available ciphers for sockets created with this context. It should be a string in the OpenSSL cipher list format. If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an SSLError will be raised.

Note: when connected, the SSLSocket.cipher() method of SSL sockets will give the currently selected cipher.

OpenSSL 1.1.1 has TLS 1.3 cipher suites enabled by default. The suites cannot be disabled with set_ciphers().

SSLContext.set_alpn_protocols(protocols)
Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of ASCII strings, like ['http/1.1', 'spdy/2'], ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to RFC 7301. After a successful handshake, the SSLSocket.selected_alpn_protocol() method will return the agreed-upon protocol.

This method will raise NotImplementedError if HAS_ALPN is False.

OpenSSL 1.1.0 to 1.1.0e will abort the handshake and raise SSLError when both sides support ALPN but cannot agree on a protocol. 1.1.0f+ behaves like 1.0.2, SSLSocket.selected_alpn_protocol() returns None.

New in version 2.7.10.

SSLContext.set_npn_protocols(protocols)
Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like ['http/1.1', 'spdy/2'], ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the NPN draft specification. After a successful handshake, the SSLSocket.selected_npn_protocol() method will return the agreed-upon protocol.

This method will raise NotImplementedError if HAS_NPN is False.

SSLContext.set_servername_callback(server_name_callback)
Register a callback function that will be called after the TLS Client Hello handshake message has been received by the SSL/TLS server when the TLS client specifies a server name indication. The server name indication mechanism is specified in RFC 6066 section 3 - Server Name Indication.

Only one callback can be set per SSLContext. If server_name_callback is None then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function, server_name_callback, will be called with three arguments; the first being the ssl.SSLSocket, the second is a string that represents the server name that the client is intending to communicate (or None if the TLS Client Hello does not contain a server name) and the third argument is the original SSLContext. The server name argument is the IDNA decoded server name.

A typical use of this callback is to change the ssl.SSLSocket's SSLSocket.context attribute to a new object of type SSLContext representing a certificate chain that matches the server name.

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like SSLSocket.selected_alpn_protocol() and SSLSocket.context. SSLSocket.getpeercert(), SSLSocket.getpeercert(), SSLSocket.cipher() and SSLSocket.compress() methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not contain return meaningful values nor can they be called safely.

The server_name_callback function must return None to allow the TLS negotiation to continue. If a TLS failure is required, a constant ALERT_DESCRIPTION_* can be returned. Other return values will result in a TLS fatal error with ALERT_DESCRIPTION_INTERNAL_ERROR.

If there is an IDNA decoding error on the server name, the TLS connection will terminate with an ALERT_DESCRIPTION_INTERNAL_ERROR fatal TLS alert message to the client.

If an exception is raised from the server_name_callback function the TLS connection will terminate with a fatal TLS alert message ALERT_DESCRIPTION_HANDSHAKE_FAILURE.

This method will raise NotImplementedError if the OpenSSL library had OPENSSL_NO_TLSEXT defined when it was built.

SSLContext.load_dh_params(dhfile)
Load the key generation parameters for Diffie-Helman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The dhfile parameter should be the path to a file containing DH parameters in PEM format.

This setting doesn't apply to client sockets. You can also use the OP_SINGLE_DH_USE option to further improve security.

SSLContext.set_ecdh_curve(curve_name)
Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The curve_name parameter should be a string describing a well-known elliptic curve, for example prime256v1 for a widely supported curve.

This setting doesn't apply to client sockets. You can also use the OP_SINGLE_ECDH_USE option to further improve security.

This method is not available if HAS_ECDH is False.

See also:

SSL/TLS & Perfect Forward Secrecy Vincent Bernat.

SSLContext.wrap_socket(sock, server_side=False, do_handshake_on_connect=True, suppress_ragged_eofs=True, server_hostname=None)
Wrap an existing Python socket sock and return an SSLSocket object. sock must be a SOCK_STREAM socket; other socket types are unsupported.

The returned SSL socket is tied to the context, its settings and certificates. The parameters server_side, do_handshake_on_connect and suppress_ragged_eofs have the same meaning as in the top-level wrap_socket() function.

On client connections, the optional parameter server_hostname specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying server_hostname will raise a ValueError if server_side is true.

Changed in version 2.7.9: Always allow a server_hostname to be passed, even if OpenSSL does not have SNI.

SSLContext.session_stats()
Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.check_hostname
Wether to match the peer cert's hostname with match_hostname() in SSLSocket.do_handshake(). The context's verify_mode must be set to CERT_OPTIONAL or CERT_REQUIRED, and you must pass server_hostname to wrap_socket() in order to match the hostname.

Example:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLS)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

Note: This features requires OpenSSL 0.9.8f or newer.

SSLContext.options
An integer representing the set of SSL options enabled on this context. The default value is OP_ALL, but you can specify other options such as OP_NO_SSLv2 by ORing them together.

Note: With versions of OpenSSL older than 0.9.8m, it is only possible to set options, not to clear them. Attempting to clear an option (by resetting the corresponding bits) will raise a ValueError.

SSLContext.protocol
The protocol version chosen when constructing the context. This attribute is read-only.

SSLContext.verify_flags
The flags for certificate verification operations. You can set flags like VERIFY_CRL_CHECK_LEAF by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs). Available only with openssl version 0.9.8+.

SSLContext.verify_mode
Whether to try to verify other peers' certificates and how to behave if verification fails. This attribute must be one of CERT_NONE, CERT_OPTIONAL or CERT_REQUIRED.

17.3.4 Certificates

Certificates in general are part of a public-key / private-key system. In this system, each principal, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the public key; the other part is kept secret, and is called the private key.

The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and only with the other part.

A certificate contains information about two principals. It contains the name of a subject, and the subject's public key. It also contains a statement by a second principal, the issuer, that the subject is who they claim to be, and that this is indeed the subject's public key. The issuer's statement is signed with the issuer's private key, which only the issuer knows. However, anyone can verify the issuer's statement by finding the issuer's public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called "notBefore" and "notAfter".

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as "PEM" (see RFC 1422), which is a base-64 encoded form wrapped with a header line and a footer line:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Certificate chains

The Python files which contain certificates can contain a sequence of certificates, sometimes called a certificate chain. This chain should start with the specific certificate for the principal who "is" the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of that certificate, and so on up the chain till you get to a certificate which is self-signed, that is, a certificate which has the same subject and issuer, sometimes called a root certificate. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority's certificate:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA certificates

If you are going to require validation of the other side of the connection's certificate, you need to provide a "CA certs" file, filled with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches. The platform's certificates file can be used by calling SSLContext.load_default_certs(), this is done automatically with create_default_context().

Combined key and certificate

Often the private key is stored in the same file as the certificate; in this case, only the certfile parameter to SSLContext.load_cert_chain() and wrap_socket() needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Self-signed certificates

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do this is with the OpenSSL package, using something like the following:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
.............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.

17.3.5 Examples

Testing for SSL support

To test for the presence of SSL support in a Python installation, user code should use the following idiom:

```
try:
    import ssl
except ImportError:
```

(continues on next page)

(continued from previous page)

```
    pass
else:
    ...  # do something that requires SSL support
```

Client-side operation

This example creates a SSL context with the recommended security settings for client sockets, including automatic certificate verification:

```
>>> context = ssl.create_default_context()
```

If you prefer to tune security settings yourself, you might create a context from scratch (but beware that you might not get the settings right):

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS)
>>> context.verify_mode = ssl.CERT_REQUIRED
>>> context.check_hostname = True
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(this snippet assumes your operating system places a bundle of all CA certificates in /etc/ssl/certs/ca-bundle.crt; if not, you'll get an error and have to adjust the location)

When you use the context to connect to a server, CERT_REQUIRED validates the server certificate: it ensures that the server certificate was signed with one of the CA certificates, and checks the signature for correctness:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

You may then fetch the certificate:

```
>>> cert = conn.getpeercert()
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host www.python.org):

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
```

(continues on next page)

(continued from previous page)

```
          (('countryName', 'US'),),
          (('stateOrProvinceName', 'NH'),),
          (('localityName', 'Wolfeboro,'),),
          (('organizationName', 'Python Software Foundation'),),
          (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.python.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

Now the SSL channel is established and the certificate verified, you can proceed to talk with the server:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

See the discussion of Security considerations below.

Server-side operation

For server operation, typically you'll need to have a server certificate, and private key, each in a file. You'll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you'll open a socket, bind it to a port, call listen() on it, and start waiting for clients to connect:

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
```

(continues on next page)

(continued from previous page)

```
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)
```

When a client connects, you'll call accept() on the socket to get the new socket from the other end, and use the context's SSLContext.wrap_socket() method to create a server-side SSL socket for the connection:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you'll read data from the connstream and do something with it till you are finished with the client (or the client is finished with you):

```
def deal_with_client(connstream):
    data = connstream.read()
    # null data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.read()
    # finished with client
```

And go back to listening for new client connections (of course, a real server would probably handle each client connection in a separate thread, or put the sockets in non-blocking mode and use an event loop).

17.3.6 Notes on non-blocking sockets

When working with non-blocking sockets, there are several things you need to be aware of:

- Calling select() tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle SSLSocket.recv() and SSLSocket.send() failures, and retry after another call to select().
- Conversely, since the SSL layer has its own framing, a SSL socket may still have data available for reading without select() being aware of it. Therefore, you should first call SSLSocket.recv() to drain any potentially available data, and then only block on a select() call if still necessary.

  (of course, similar provisions apply when using other primitives such as poll(), or those in the selectors module)
- The SSL handshake itself will be non-blocking: the SSLSocket.do_handshake() method has to be retried until it returns successfully. Here is a synopsis using select() to wait for the socket's readiness:

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

17.3.7 Security considerations

Best defaults

For client use, if you don't have any special requirements for your security policy, it is highly recommended that you use the create_default_context() function to create your SSL context. It will load the system's trusted CA certificates, enable certificate validation and hostname checking, and try to choose reasonably secure protocol and cipher settings.

If a client certificate is needed for the connection, it can be added with SSLContext.load_cert_chain().

By contrast, if you create the SSL context by calling the SSLContext constructor yourself, it will not have certificate validation nor hostname checking enabled by default. If you do so, please read the paragraphs below to achieve a good security level.

Manual settings

Verifying certificates

When calling the SSLContext constructor directly, CERT_NONE is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of time you would like to ensure the authenticity of the server you're talking to. Therefore, when in client mode, it is highly recommended to use CERT_REQUIRED. However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling SSLSocket.getpeercert(), matches the desired service. For many protocols and applications, the service can be identified by the hostname; in this case, the match_hostname() function can be used. This common check is automatically performed when SSLContext.check_hostname is enabled.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you'll also have to specify CERT_REQUIRED and similarly check the client certificate.

> Note: In client mode, CERT_OPTIONAL and CERT_REQUIRED are equivalent unless anonymous ciphers are enabled (they are disabled by default).

Protocol versions

SSL versions 2 and 3 are considered insecure and are therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use PROTOCOL_SSLv23 as the protocol version and then disable SSLv2 and SSLv3 explicitly using the SSLContext.options attribute:

```
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
context.options |= ssl.OP_NO_SSLv2
context.options |= ssl.OP_NO_SSLv3
```

The SSL context created above will only allow TLSv1 and later (if supported by your system) connections.

Cipher selection

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the SSLContext.set_ciphers() method. Starting from Python 2.7.9, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL's documentation about the cipher list format. If you want to check which ciphers are enabled by a given cipher list, use the openssl ciphers command on your system.

Multi-processing

If using this module as part of a multi-processed application (using, for example the multiprocessing or concurrent.futures modules), be aware that OpenSSL's internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with os.fork(). Any successful call of RAND_add(), RAND_bytes() or RAND_pseudo_bytes() is sufficient.

17.3.8 LibreSSL support

LibreSSL is a fork of OpenSSL 1.0.1. The ssl module has limited support for LibreSSL. Some features are not available when the ssl module is compiled with LibreSSL.

- LibreSSL >= 2.6.1 no longer supports NPN. The methods SSLContext.set_npn_protocols() and SSLSocket.selected_npn_protocol() are not available.
- SSLContext.set_default_verify_paths() ignores the env vars SSL_CERT_FILE and SSL_CERT_PATH although get_default_verify_paths() still reports them.

See also:

Class socket.socket Documentation of underlying socket class

SSL/TLS Strong Encryption: An Introduction Intro from the Apache webserver documentation

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management Steve Kent

RFC 1750: Randomness Recommendations for Security D. Eastlake et. al.

RFC 3280: Internet X.509 Public Key Infrastructure Certificate and CRL Profile Housley et. al.

RFC 4366: Transport Layer Security (TLS) Extensions Blake-Wilson et. al.

RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2 T. Dierks et. al.

RFC 6066: Transport Layer Security (TLS) Extensions D. Eastlake

IANA TLS: Transport Layer Security (TLS) Parameters IANA

RFC 7525: Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (I
IETF

Mozilla's Server Side TLS recommendations Mozilla

17.4 signal — Set handlers for asynchronous events

This module provides mechanisms to use signal handlers in Python. Some general rules for working with signals and their handlers:

- A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for SIGCHLD, which follows the underlying implementation.
- There is no way to "block" signals temporarily from critical sections (since this is not supported by all Unix flavors).
- Although Python signal handlers are called asynchronously as far as the Python user is concerned, they can only occur between the "atomic" instructions of the Python interpreter. This means that signals arriving during long calculations implemented purely in C (such as regular expression matches on large bodies of text) may be delayed for an arbitrary amount of time.
- When a signal arrives during an I/O operation, it is possible that the I/O operation raises an exception after the signal handler returns. This is dependent on the underlying Unix system's semantics regarding interrupted system calls.
- Because the C signal handler always returns, it makes little sense to catch synchronous errors like SIGFPE or SIGSEGV.
- Python installs a small number of signal handlers by default: SIGPIPE is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and SIGINT is translated into a KeyboardInterrupt exception. All of these can be overridden.
- Some care must be taken if both signals and threads are used in the same program. The fundamental thing to remember in using signals and threads simultaneously is: always perform signal() operations in the main thread of execution. Any thread can perform an alarm(), getsignal(), pause(), setitimer() or getitimer(); only the main thread can set a new signal handler, and the main thread will be the only one to receive signals (this is enforced by the Python signal module, even if the underlying thread implementation supports sending signals to individual threads). This means that signals can't be used as a means of inter-thread communication. Use locks instead.

The variables defined in the signal module are:

signal.SIG_DFL
: This is one of two standard signal handling options; it will simply perform the default function for the signal. For example, on most systems the default action for SIGQUIT is to dump core and exit, while the default action for SIGCHLD is to simply ignore it.

signal.SIG_IGN
: This is another standard signal handler, which will simply ignore the given signal.

SIG*
: All the signal numbers are defined symbolically. For example, the hangup signal is defined as signal.SIGHUP; the variable names are identical to the names used in C programs, as found in <signal.h>. The Unix man page for 'signal()' lists the existing signals (on some systems this is signal(2), on others the list is in signal(7)). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

signal.CTRL_C_EVENT
: The signal corresponding to the Ctrl+C keystroke event. This signal can only be used with os.kill().

  Availability: Windows.

  New in version 2.7.

signal.CTRL_BREAK_EVENT
The signal corresponding to the Ctrl+Break keystroke event. This signal can only be used with os.kill().

Availability: Windows.

New in version 2.7.

signal.NSIG
One more than the number of the highest signal number.

signal.ITIMER_REAL
Decrements interval timer in real time, and delivers SIGALRM upon expiration.

signal.ITIMER_VIRTUAL
Decrements interval timer only when the process is executing, and delivers SIGVTALRM upon expiration.

signal.ITIMER_PROF
Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

The signal module defines one exception:

exception signal.ItimerError
Raised to signal an error from the underlying setitimer() or getitimer() implementation. Expect this error if an invalid interval timer or a negative time is passed to setitimer(). This error is a subtype of IOError.

The signal module defines the following functions:

signal.alarm(time)
If time is non-zero, this function requests that a SIGALRM signal be sent to the process in time seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If time is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled. (See the Unix man page alarm(2).) Availability: Unix.

signal.getsignal(signalnum)
Return the current signal handler for the signal signalnum. The returned value may be a callable Python object, or one of the special values signal.SIG_IGN, signal.SIG_DFL or None. Here, signal.SIG_IGN means that the signal was previously ignored, signal.SIG_DFL means that the default way of handling the signal was previously in use, and None means that the previous signal handler was not installed from Python.

signal.pause()
Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing. Not on Windows. (See the Unix man page signal(2).)

signal.setitimer(which, seconds[, interval])
Sets given interval timer (one of signal.ITIMER_REAL, signal.ITIMER_VIRTUAL or signal.ITIMER_PROF) specified by which to fire after seconds (float is accepted, different from alarm()) and after that every interval seconds. The interval timer specified by which can be cleared by setting seconds to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used; signal.ITIMER_REAL will deliver SIGALRM, signal.ITIMER_VIRTUAL sends SIGVTALRM, and signal.ITIMER_PROF will deliver SIGPROF.

The old values are returned as a tuple: (delay, interval).

Attempting to pass an invalid interval timer will cause an ItimerError. Availability: Unix.

New in version 2.6.

signal.getitimer(which)
Returns current value of a given interval timer specified by which. Availability: Unix.

New in version 2.6.

signal.set_wakeup_fd(fd)
Set the wakeup fd to fd. When a signal is received, a '\0' byte is written to the fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned (or -1 if file descriptor wakeup was not enabled). If fd is -1, file descriptor wakeup is disabled. If not -1, fd must be non-blocking. It is up to the library to remove any bytes from fd before calling poll or select again.

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a ValueError exception to be raised.

New in version 2.6.

signal.siginterrupt(signalnum, flag)
Change system call restart behaviour: if flag is False, system calls will be restarted when interrupted by signal signalnum, otherwise system calls will be interrupted. Returns nothing. Availability: Unix (see the man page siginterrupt(3) for further information).

Note that installing a signal handler with signal() will reset the restart behaviour to interruptible by implicitly calling siginterrupt() with a true flag value for the given signal.

New in version 2.6.

signal.signal(signalnum, handler)
Set the handler for signal signalnum to the function handler. handler can be a callable Python object taking two arguments (see below), or one of the special values signal.SIG_IGN or signal.SIG_DFL. The previous signal handler will be returned (see the description of getsignal() above). (See the Unix man page signal(2).)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a ValueError exception to be raised.

The handler is called with two arguments: the signal number and the current stack frame (None or a frame object; for a description of frame objects, see the description in the type hierarchy or see the attribute descriptions in the inspect module).

On Windows, signal() can only be called with SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, or SIGTERM. A ValueError will be raised in any other case.

17.4.1 Example

Here is a minimal example program. It uses the alarm() function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the os.open() to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```
import signal, os

def handler(signum, frame):
    print 'Signal handler called with signal', signum
    raise IOError("Couldn't open device!")
```

(continues on next page)

(continued from previous page)

```
# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

17.5 popen2 — Subprocesses with accessible I/O streams

Deprecated since version 2.6: This module is obsolete. Use the subprocess module. Check especially the Replacing Older Functions with the subprocess Module section.

This module allows you to spawn processes and connect to their input/output/error pipes and obtain their return codes under Unix and Windows.

The subprocess module provides more powerful facilities for spawning new processes and retrieving their results. Using the subprocess module is preferable to using the popen2 module.

The primary interface offered by this module is a trio of factory functions. For each of these, if bufsize is specified, it specifies the buffer size for the I/O pipes. mode, if provided, should be the string `'b'` or `'t'`; on Windows this is needed to determine whether the file objects should be opened in binary or text mode. The default value for mode is `'t'`.

On Unix, cmd may be a sequence, in which case arguments will be passed directly to the program without shell intervention (as with os.spawnv()). If cmd is a string it will be passed to the shell (as with os.system()).

The only way to retrieve the return codes for the child processes is by using the poll() or wait() methods on the Popen3 and Popen4 classes; these are only available on Unix. This information is not available when using the popen2(), popen3(), and popen4() functions, or the equivalent functions in the os module. (Note that the tuples returned by the os module's functions are in a different order from the ones returned by the popen2 module.)

popen2.popen2(cmd[, bufsize[, mode]])
: Executes cmd as a sub-process. Returns the file objects (child_stdout, child_stdin).

popen2.popen3(cmd[, bufsize[, mode]])
: Executes cmd as a sub-process. Returns the file objects (child_stdout, child_stdin, child_stderr).

popen2.popen4(cmd[, bufsize[, mode]])
: Executes cmd as a sub-process. Returns the file objects (child_stdout_and_stderr, child_stdin).

  New in version 2.0.

On Unix, a class defining the objects returned by the factory functions is also available. These are not used for the Windows implementation, and are not available on that platform.

class popen2.Popen3(cmd[, capturestderr[, bufsize]])
: This class represents a child process. Normally, Popen3 instances are created using the popen2() and popen3() factory functions described above.

  If not using one of the helper functions to create Popen3 objects, the parameter cmd is the shell command to execute in a sub-process. The capturestderr flag, if true, specifies that the object should capture standard error output of the child process. The default is false. If the bufsize parameter is specified, it specifies the size of the I/O buffers to/from the child process.

class popen2.Popen4(cmd[, bufsize])
Similar to Popen3, but always captures standard error into the same file object as standard output. These are typically created using popen4().

New in version 2.0.

17.5.1 Popen3 and Popen4 Objects

Instances of the Popen3 and Popen4 classes have the following methods:

Popen3.poll()
Returns -1 if child process hasn't completed yet, or its status code (see wait()) otherwise.

Popen3.wait()
Waits for and returns the status code of the child process. The status code encodes both the return code of the process and information about whether it exited using the exit() system call or died due to a signal. Functions to help interpret the status code are defined in the os module; see section Process Management for the W*() family of functions.

The following attributes are also available:

Popen3.fromchild
A file object that provides output from the child process. For Popen4 instances, this will provide both the standard output and standard error streams.

Popen3.tochild
A file object that provides input to the child process.

Popen3.childerr
A file object that provides error output from the child process, if capturestderr was true for the constructor, otherwise None. This will always be None for Popen4 instances.

Popen3.pid
The process ID of the child process.

17.5.2 Flow Control Issues

Any time you are working with any form of inter-process communication, control flow needs to be carefully thought out. This remains the case with the file objects provided by this module (or the os module equivalents).

When reading output from a child process that writes a lot of data to standard error while the parent is reading from the child's standard output, a deadlock can occur. A similar situation can occur with other combinations of reads and writes. The essential factors are that more than _PC_PIPE_BUF bytes are being written by one process in a blocking fashion, while the other process is reading from the first process, also in a blocking fashion.

There are several ways to deal with this situation.

The simplest application change, in many cases, will be to follow this model in the parent process:

```
import popen2

r, w, e = popen2.popen3('python slave.py')
e.readlines()
r.readlines()
r.close()
```

(continues on next page)

(continued from previous page)

```
e.close()
w.close()
```

with code like this in the child:

```
import os
import sys

# note that each of these print statements
# writes a single long string

print >>sys.stderr, 400 * 'this is a test\n'
os.close(sys.stderr.fileno())
print >>sys.stdout, 400 * 'this is another test\n'
```

In particular, note that sys.stderr must be closed after writing all data, or readlines() won't return. Also note that os.close() must be used, as sys.stderr.close() won't close stderr (otherwise assigning to sys.stderr will silently close it, so no further errors can be printed).

Applications which need to support a more general approach should integrate I/O over pipes with their select() loops, or use separate threads to read each of the individual files provided by whichever popen*() function or Popen* class was used.

See also:

Module subprocess Module for spawning and managing subprocesses.

17.6 asyncore — Asynchronous socket handler

Source code: Lib/asyncore.py

This module provides the basic infrastructure for writing asynchronous socket service clients and servers.

There are only two ways to have a program on a single processor do "more than one thing at a time." Multi-threaded programming is the simplest and most popular way to do it, but there is another very different technique, that lets you have nearly all the advantages of multi-threading, without actually using multiple threads. It's really only practical if your program is largely I/O bound. If your program is processor bound, then pre-emptive scheduled threads are probably what you really need. Network servers are rarely processor bound, however.

If your operating system supports the select() system call in its I/O library (and nearly all do), then you can use it to juggle multiple communication channels at once; doing other work while your I/O is taking place in the "background." Although this strategy can seem strange and complex, especially at first, it is in many ways easier to understand and control than multi-threaded programming. The asyncore module solves many of the difficult problems for you, making the task of building sophisticated high-performance network servers and clients a snap. For "conversational" applications and protocols the companion asynchat module is invaluable.

The basic idea behind both modules is to create one or more network channels, instances of class asyncore.dispatcher and asynchat.async_chat. Creating the channels adds them to a global map, used by the loop() function if you do not provide it with your own map.

Once the initial channel(s) is(are) created, calling the loop() function activates channel service, which continues until the last channel (including any that have been added to the map during asynchronous service) is closed.

asyncore.loop([timeout[, use_poll[, map[, count]]]])
Enter a polling loop that terminates after count passes or all open channels have been closed. All arguments are optional. The count parameter defaults to None, resulting in the loop terminating only when all channels have been closed. The timeout argument sets the timeout parameter for the appropriate select() or poll() call, measured in seconds; the default is 30 seconds. The use_poll parameter, if true, indicates that poll() should be used in preference to select() (the default is False).

The map parameter is a dictionary whose items are the channels to watch. As channels are closed they are deleted from their map. If map is omitted, a global map is used. Channels (instances of asyncore.dispatcher, asynchat.async_chat and subclasses thereof) can freely be mixed in the map.

class asyncore.dispatcher
The dispatcher class is a thin wrapper around a low-level socket object. To make it more useful, it has a few methods for event-handling which are called from the asynchronous loop. Otherwise, it can be treated as a normal non-blocking socket object.

The firing of low-level events at certain times or in certain connection states tells the asynchronous loop that certain higher-level events have taken place. For example, if we have asked for a socket to connect to another host, we know that the connection has been made when the socket becomes writable for the first time (at this point you know that you may write to it with the expectation of success). The implied higher-level events are:

Event	Description
handle_connect()	Implied by the first read or write event
handle_close()	Implied by a read event with no data available
handle_accept()	Implied by a read event on a listening socket

During asynchronous processing, each mapped channel's readable() and writable() methods are used to determine whether the channel's socket should be added to the list of channels select()ed or poll()ed for read and write events.

Thus, the set of channel events is larger than the basic socket events. The full set of methods that can be overridden in your subclass follows:

handle_read()
Called when the asynchronous loop detects that a read() call on the channel's socket will succeed.

handle_write()
Called when the asynchronous loop detects that a writable socket can be written. Often this method will implement the necessary buffering for performance. For example:

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

handle_expt()
Called when there is out of band (OOB) data for a socket connection. This will almost never happen, as OOB is tenuously supported and rarely used.

handle_connect()
Called when the active opener's socket actually makes a connection. Might send a "welcome" banner, or initiate a protocol negotiation with the remote endpoint, for example.

handle_close()
Called when the socket is closed.

handle_error()
Called when an exception is raised and not otherwise handled. The default version prints a

condensed traceback.

handle_accept()
: Called on listening channels (passive openers) when a connection can be established with a new remote endpoint that has issued a connect() call for the local endpoint.

readable()
: Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which read events can occur. The default method simply returns True, indicating that by default, all channels will be interested in read events.

writable()
: Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which write events can occur. The default method simply returns True, indicating that by default, all channels will be interested in write events.

In addition, each channel delegates or extends many of the socket methods. Most of these are nearly identical to their socket partners.

create_socket(family, type)
: This is identical to the creation of a normal socket, and will use the same options for creation. Refer to the socket documentation for information on creating sockets.

connect(address)
: As with the normal socket object, address is a tuple with the first element the host to connect to, and the second the port number.

send(data)
: Send data to the remote end-point of the socket.

recv(buffer_size)
: Read at most buffer_size bytes from the socket's remote end-point. An empty string implies that the channel has been closed from the other end.

  Note that recv() may raise socket.error with EAGAIN or EWOULDBLOCK, even though select.select() or select.poll() has reported the socket ready for reading.

listen(backlog)
: Listen for connections made to the socket. The backlog argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

bind(address)
: Bind the socket to address. The socket must not already be bound. (The format of address depends on the address family — refer to the socket documentation for more information.) To mark the socket as re-usable (setting the SO_REUSEADDR option), call the dispatcher object's set_reuse_addr() method.

accept()
: Accept a connection. The socket must be bound to an address and listening for connections. The return value can be either None or a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection. When None is returned it means the connection didn't take place, in which case the server should just ignore this event and keep listening for further incoming connections.

close()
: Close the socket. All future operations on the socket object will fail. The remote end-point will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

class asyncore.dispatcher_with_send
: A dispatcher subclass which adds simple buffered output capability, useful for simple clients. For more sophisticated usage use asynchat.async_chat.

class asyncore.file_dispatcher
: A file_dispatcher takes a file descriptor or file object along with an optional map argument and wraps it for use with the poll() or loop() functions. If provided a file object or anything with a fileno() method, that method will be called and passed to the file_wrapper constructor. Availability: UNIX.

class asyncore.file_wrapper
: A file_wrapper takes an integer file descriptor and calls os.dup() to duplicate the handle so that the original handle may be closed independently of the file_wrapper. This class implements sufficient methods to emulate a socket for use by the file_dispatcher class. Availability: UNIX.

17.6.1 asyncore Example basic HTTP client

Here is a very basic HTTP client that uses the dispatcher class to implement its socket handling:

```
import asyncore, socket

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect( (host, 80) )
        self.buffer = 'GET %s HTTP/1.0\r\n\r\n' % path

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print self.recv(8192)

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]


client = HTTPClient('www.python.org', '/')
asyncore.loop()
```

17.6.2 asyncore Example basic echo server

Here is a basic echo server that uses the dispatcher class to accept connections and dispatches the incoming connections to a handler:

```
import asyncore
import socket
```

(continues on next page)

(continued from previous page)

```
class EchoHandler(asyncore.dispatcher_with_send):

    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):

    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)

    def handle_accept(self):
        pair = self.accept()
        if pair is not None:
            sock, addr = pair
            print 'Incoming connection from %s' % repr(addr)
            handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```

17.7 asynchat — Asynchronous socket command/response handler

Source code: Lib/asynchat.py

This module builds on the asyncore infrastructure, simplifying asynchronous clients and servers and making it easier to handle protocols whose elements are terminated by arbitrary strings, or are of variable length. asynchat defines the abstract class async_chat that you subclass, providing implementations of the collect_incoming_data() and found_terminator() methods. It uses the same asynchronous loop as asyncore, and the two types of channel, asyncore.dispatcher and asynchat.async_chat, can freely be mixed in the channel map. Typically an asyncore.dispatcher server channel generates new asynchat.async_chat channel objects as it receives incoming connection requests.

class asynchat.async_chat
: This class is an abstract subclass of asyncore.dispatcher. To make practical use of the code you must subclass async_chat, providing meaningful collect_incoming_data() and found_terminator() methods. The asyncore.dispatcher methods can be used, although not all make sense in a message/response context.

  Like asyncore.dispatcher, async_chat defines a set of events that are generated by an analysis of socket conditions after a select() call. Once the polling loop has been started the async_chat object's methods are called by the event-processing framework with no action on the part of the programmer.

  Two class attributes can be modified, to improve performance, or possibly even to conserve memory.

  ac_in_buffer_size
  : The asynchronous input buffer size (default 4096).

ac_out_buffer_size
: The asynchronous output buffer size (default 4096).

Unlike asyncore.dispatcher, async_chat allows you to define a first-in-first-out queue (fifo) of producers. A producer need have only one method, more(), which should return data to be transmitted on the channel. The producer indicates exhaustion (i.e. that it contains no more data) by having its more() method return the empty string. At this point the async_chat object removes the producer from the fifo and starts using the next producer, if any. When the producer fifo is empty the handle_write() method does nothing. You use the channel object's set_terminator() method to describe how to recognize the end of, or an important breakpoint in, an incoming transmission from the remote endpoint.

To build a functioning async_chat subclass your input methods collect_incoming_data() and found_terminator() must handle the data that the channel receives asynchronously. The methods are described below.

async_chat.close_when_done()
: Pushes a None on to the producer fifo. When this producer is popped off the fifo it causes the channel to be closed.

async_chat.collect_incoming_data(data)
: Called with data holding an arbitrary amount of received data. The default method, which must be overridden, raises a NotImplementedError exception.

async_chat.discard_buffers()
: In emergencies this method will discard any data held in the input and/or output buffers and the producer fifo.

async_chat.found_terminator()
: Called when the incoming data stream matches the termination condition set by set_terminator(). The default method, which must be overridden, raises a NotImplementedError exception. The buffered input data should be available via an instance attribute.

async_chat.get_terminator()
: Returns the current terminator for the channel.

async_chat.push(data)
: Pushes data on to the channel's fifo to ensure its transmission. This is all you need to do to have the channel write the data out to the network, although it is possible to use your own producers in more complex schemes to implement encryption and chunking, for example.

async_chat.push_with_producer(producer)
: Takes a producer object and adds it to the producer fifo associated with the channel. When all currently-pushed producers have been exhausted the channel will consume this producer's data by calling its more() method and send the data to the remote endpoint.

async_chat.set_terminator(term)
: Sets the terminating condition to be recognized on the channel. term may be any of three types of value, corresponding to three different ways to handle incoming protocol data.

term	Description
string	Will call found_terminator() when the string is found in the input stream
integer	Will call found_terminator() when the indicated number of characters have been received
None	The channel continues to collect data forever

Note that any data following the terminator will be available for reading by the channel after found_terminator() is called.

17.7.1 asynchat - Auxiliary Classes

class asynchat.fifo([list=None])
: A fifo holding data which has been pushed by the application but not yet popped for writing to the channel. A fifo is a list used to hold data and/or producers until they are required. If the list argument is provided then it should contain producers or data items to be written to the channel.

  is_empty()
  : Returns True if and only if the fifo is empty.

  first()
  : Returns the least-recently push()ed item from the fifo.

  push(data)
  : Adds the given data (which may be a string or a producer object) to the producer fifo.

  pop()
  : If the fifo is not empty, returns True, first(), deleting the popped item. Returns False, None for an empty fifo.

17.7.2 asynchat Example

The following partial example shows how HTTP requests can be read with async_chat. A web server might create an http_request_handler object for each incoming client connection. Notice that initially the channel terminator is set to match the blank line at the end of the HTTP headers, and a flag indicates that the headers are being read.

Once the headers have been read, if the request is of type POST (indicating that further data are present in the input stream) then the Content-Length: header is used to set a numeric terminator to read the right amount of data from the channel.

The handle_request() method is called once all relevant input has been marshalled, after setting the channel terminator to None to ensure that any extraneous data sent by the web client are ignored.

```
class http_request_handler(asynchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = ""
        self.set_terminator("\r\n\r\n")
        self.reading_headers = True
        self.handling = False
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, data):
        """Buffer the data"""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
            self.reading_headers = False
            self.parse_headers("".join(self.ibuffer))
            self.ibuffer = []
```

(continues on next page)

(continued from previous page)

```
            if self.op.upper() == "POST":
                clen = self.headers.getheader("content-length")
                self.set_terminator(int(clen))
            else:
                self.handling = True
                self.set_terminator(None)
                self.handle_request()
        elif not self.handling:
            self.set_terminator(None)  # browsers sometimes over-send
            self.cgi_data = parse(self.headers, "".join(self.ibuffer))
            self.handling = True
            self.ibuffer = []
            self.handle_request()
```

CHAPTER

EIGHTEEN

INTERNET DATA HANDLING

This chapter describes modules which support handling data formats commonly used on the Internet.

18.1 email — An email and MIME handling package

New in version 2.2.

The email package is a library for managing email messages, including MIME and other RFC 2822-based message documents. It subsumes most of the functionality in several older standard modules such as rfc822, mimetools, multifile, and other non-standard packages such as mimecntl. It is specifically not designed to do any sending of email messages to SMTP (RFC 2821), NNTP, or other servers; those are functions of modules such as smtplib and nntplib. The email package attempts to be as RFC-compliant as possible, supporting in addition to RFC 2822, such MIME-related RFCs as RFC 2045, RFC 2046, RFC 2047, and RFC 2231.

The primary distinguishing feature of the email package is that it splits the parsing and generating of email messages from the internal object model representation of email. Applications using the email package deal primarily with objects; you can add sub-objects to messages, remove sub-objects from messages, completely re-arrange the contents, etc. There is a separate parser and a separate generator which handles the transformation from flat text to the object model, and then back to flat text again. There are also handy subclasses for some common MIME object types, and a few miscellaneous utilities that help with such common tasks as extracting and parsing message field values, creating RFC-compliant dates, etc.

The following sections describe the functionality of the email package. The ordering follows a progression that should be common in applications: an email message is read as flat text from a file or other source, the text is parsed to produce the object structure of the email message, this structure is manipulated, and finally, the object tree is rendered back into flat text.

It is perfectly feasible to create the object structure out of whole cloth — i.e. completely from scratch. From there, a similar progression can be taken as above.

Also included are detailed specifications of all the classes and modules that the email package provides, the exception classes you might encounter while using the email package, some auxiliary utilities, and a few examples. For users of the older mimelib package, or previous versions of the email package, a section on differences and porting is provided.

Contents of the email package documentation:

18.1.1 email.message: Representing an email message

The central class in the email package is the Message class, imported from the email.message module. It is the base class for the email object model. Message provides the core functionality for setting and querying header fields, and for accessing message bodies.

Conceptually, a Message object consists of headers and payloads. Headers are RFC 2822 style field names and values where the field name and value are separated by a colon. The colon is not part of either the field name or the field value.

Headers are stored and returned in case-preserving form but are matched case-insensitively. There may also be a single envelope header, also known as the Unix-From header or the From_ header. The payload is either a string in the case of simple message objects or a list of Message objects for MIME container documents (e.g. multipart/* and message/rfc822).

Message objects provide a mapping style interface for accessing the message headers, and an explicit interface for accessing both the headers and the payload. It provides convenience methods for generating a flat text representation of the message object tree, for accessing commonly used header parameters, and for recursively walking over the object tree.

Here are the methods of the Message class:

class email.message.Message
The constructor takes no arguments.

as_string([unixfrom])
Return the entire message flattened as a string. When optional unixfrom is True, the envelope header is included in the returned string. unixfrom defaults to False. Flattening the message may trigger changes to the Message if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it mangles lines that begin with From. For more flexibility, instantiate a Generator instance and use its flatten() method directly. For example:

```
from cStringIO import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=False, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

__str__()
Equivalent to as_string(unixfrom=True).

is_multipart()
Return True if the message's payload is a list of sub-Message objects, otherwise return False. When is_multipart() returns False, the payload should be a string object.

set_unixfrom(unixfrom)
Set the message's envelope header to unixfrom, which should be a string.

get_unixfrom()
Return the message's envelope header. Defaults to None if the envelope header was never set.

attach(payload)
Add the given payload to the current payload, which must be None or a list of Message objects before the call. After the call, the payload will always be a list of Message objects. If you want to set the payload to a scalar object (e.g. a string), use set_payload() instead.

get_payload([i[, decode]])
Return the current payload, which will be a list of Message objects when is_multipart() is True, or a string when is_multipart() is False. If the payload is a list and you mutate the list object, you modify the message's payload in place.

With optional argument i, get_payload() will return the i-th element of the payload, counting from zero, if is_multipart() is True. An IndexError will be raised if i is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. is_multipart() is False) and i is given, a TypeError is raised.

Optional decode is a flag indicating whether the payload should be decoded or not, according to the Content-Transfer-Encoding header. When True and the message is not a multipart, the payload will be decoded if this header's value is quoted-printable or base64. If some other encoding is used, or Content-Transfer-Encoding header is missing, or if the payload has bogus base64 data, the payload is returned as-is (undecoded). If the message is a multipart and the decode flag is True, then None is returned. The default for decode is False.

set_payload(payload[, charset])
Set the entire message object's payload to payload. It is the client's responsibility to ensure the payload invariants. Optional charset sets the message's default character set; see set_charset() for details.

Changed in version 2.2.2: charset argument added.

set_charset(charset)
Set the character set of the payload to charset, which can either be a Charset instance (see email.charset), a string naming a character set, or None. If it is a string, it will be converted to a Charset instance. If charset is None, the charset parameter will be removed from the Content-Type header (the message will not be otherwise modified). Anything else will generate a TypeError.

If there is no existing MIME-Version header one will be added. If there is no existing Content-Type header, one will be added with a value of text/plain. Whether the Content-Type header already exists or not, its charset parameter will be set to charset.output_charset. If charset.input_charset and charset.output_charset differ, the payload will be re-encoded to the output_charset. If there is no existing Content-Transfer-Encoding header, then the payload will be transfer-encoded, if needed, using the specified Charset, and a header with the appropriate value will be added. If a Content-Transfer-Encoding header already exists, the payload is assumed to already be correctly encoded using that Content-Transfer-Encoding and is not modified.

The message will be assumed to be of type text/*, with the payload either in unicode or encoded with charset.input_charset. It will be encoded or converted to charset.output_charset and transfer encoded properly, if needed, when generating the plain text representation of the message. MIME headers (MIME-Version, Content-Type, Content-Transfer-Encoding) will be added as needed.

New in version 2.2.2.

get_charset()
Return the Charset instance associated with the message's payload.

New in version 2.2.2.

The following methods implement a mapping-like interface for accessing the message's RFC 2822 headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by keys(), but in a Message object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

__len__()
Return the total number of headers, including duplicates.

__contains__(name)
Return true if the message object has a field named name. Matching is done case-insensitively and name should not include the trailing colon. Used for the in operator, e.g.:

```
if 'message-id' in myMessage:
   print 'Message-ID:', myMessage['message-id']
```

__getitem__(name)
Return the value of the named header field. name should not include the colon field separator. If the header is missing, None is returned; a KeyError is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the get_all() method to get the values of all the extant named headers.

__setitem__(name, val)
Add a header to the message with field name name and value val. The field is appended to the end of the message's existing fields.

Note that this does not overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name name, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(name)
Delete all occurrences of the field with name name from the message's headers. No exception is raised if the named field isn't present in the headers.

has_key(name)
Return true if the message contains a header field named name, otherwise return false.

keys()
Return a list of all the message's header field names.

values()
Return a list of all the message's field values.

items()
Return a list of 2-tuples containing all the message's field headers and values.

get(name[, failobj])
Return the value of the named header field. This is identical to __getitem__() except that optional failobj is returned if the named header is missing (defaults to None).

Here are some additional useful methods:

get_all(name[, failobj])
Return a list of all the values for the field named name. If there are no such named headers in the message, failobj is returned (defaults to None).

add_header(_name, _value, **_params)
Extended header setting. This method is similar to __setitem__() except that additional header parameters can be provided as keyword arguments. _name is the header field to add and _value is the primary value for the header.

For each item in the keyword argument dictionary _params, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as key="value" unless the value is None, in which case only the key will be added. If the value contains non-ASCII characters, it must be specified as a three tuple in the format (CHARSET, LANGUAGE, VALUE), where CHARSET is a string naming the charset to be used to encode the value, LANGUAGE can usually be set to None or the empty string (see RFC 2231 for other possibilities), and VALUE is the string value containing non-ASCII code points.

Here's an example:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

Which produces

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header(_name, _value)
Replace a header. Replace the first header found in the message that matches _name, retaining header order and field name case. If no matching header was found, a KeyError is raised.

New in version 2.2.2.

get_content_type()
Return the message's content type. The returned string is coerced to lower case of the form maintype/subtype. If there was no Content-Type header in the message the default type as given by get_default_type() will be returned. Since according to RFC 2045, messages always have a default type, get_content_type() will always return a value.

RFC 2045 defines a message's default type to be text/plain unless it appears inside a multipart/digest container, in which case it would be message/rfc822. If the Content-Type header has an invalid type specification, RFC 2045 mandates that the default type be text/plain.

New in version 2.2.2.

get_content_maintype()
Return the message's main content type. This is the maintype part of the string returned by get_content_type().

New in version 2.2.2.

get_content_subtype()
Return the message's sub-content type. This is the subtype part of the string returned by get_content_type().

New in version 2.2.2.

get_default_type()
Return the default content type. Most messages have a default content type of text/plain, except for messages that are subparts of multipart/digest containers. Such subparts have a default content type of message/rfc822.

New in version 2.2.2.

set_default_type(ctype)
Set the default content type. ctype should either be text/plain or message/rfc822, although this is not enforced. The default content type is not stored in the Content-Type header.

New in version 2.2.2.

get_params([failobj[, header[, unquote]]])
Return the message's Content-Type parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the '=' sign. The left hand side of the '=' is the key, while the right hand side is the value. If there is no '=' sign in the parameter the value is the empty string, otherwise the value is as described in get_param() and is unquoted if optional unquote is True (the default).

Optional failobj is the object to return if there is no Content-Type header. Optional header is the header to search instead of Content-Type.

Changed in version 2.2.2: unquote argument added.

get_param(param[, failobj[, header[, unquote]]])
Return the value of the Content-Type header's parameter param as a string. If the message has no Content-Type header or if there is no such parameter, then failobj is returned (defaults to None).

Optional header if given, specifies the message header to use instead of Content-Type.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was RFC 2231 encoded. When it's a 3-tuple, the elements of the value are of the form (CHARSET, LANGUAGE, VALUE). Note that both CHARSET and LANGUAGE can be None, in which case you should consider VALUE to be encoded in the us-ascii charset. You can usually ignore LANGUAGE.

If your application doesn't care whether the parameter was encoded as in RFC 2231, you can collapse the parameter value by calling email.utils.collapse_rfc2231_value(), passing in the return value from get_param(). This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn't. For example:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

In any case, the parameter value (either the returned string, or the VALUE item in the 3-tuple) is always unquoted, unless unquote is set to False.

Changed in version 2.2.2: unquote argument added, and 3-tuple return value possible.

set_param(param, value[, header[, requote[, charset[, language]]]])
Set a parameter in the Content-Type header. If the parameter already exists in the header, its value will be replaced with value. If the Content-Type header as not yet been defined for this message, it will be set to text/plain and the new parameter value will be appended as per RFC 2045.

Optional header specifies an alternative header to Content-Type, and all parameters will be quoted as necessary unless optional requote is False (the default is True).

If optional charset is specified, the parameter will be encoded according to RFC 2231. Optional language specifies the RFC 2231 language, defaulting to the empty string. Both charset and language should be strings.

New in version 2.2.2.

del_param(param[, header[, requote]])
Remove the given parameter completely from the Content-Type header. The header will be rewritten in place without the parameter or its value. All values will be quoted as necessary unless requote is False (the default is True). Optional header specifies an alternative to Content-Type.

New in version 2.2.2.

set_type(type[, header][, requote])
Set the main type and subtype for the Content-Type header. type must be a string in the form maintype/subtype, otherwise a ValueError is raised.

This method replaces the Content-Type header, keeping all the parameters in place. If requote is False, this leaves the existing header's quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the header argument. When the Content-Type header is set a MIME-Version header is also added.

New in version 2.2.2.

get_filename([failobj])
Return the value of the filename parameter of the Content-Disposition header of the message. If the header does not have a filename parameter, this method falls back to looking for the name parameter on the Content-Type header. If neither is found, or the header is missing, then failobj is returned. The returned string will always be unquoted as per email.utils.unquote().

get_boundary([failobj])
Return the value of the boundary parameter of the Content-Type header of the message, or failobj if either the header is missing, or has no boundary parameter. The returned string will always be unquoted as per email.utils.unquote().

set_boundary(boundary)
Set the boundary parameter of the Content-Type header to boundary. set_boundary() will always quote boundary if necessary. A HeaderParseError is raised if the message object has no Content-Type header.

Note that using this method is subtly different than deleting the old Content-Type header and adding a new one with the new boundary via add_header(), because set_boundary() preserves the order of the Content-Type header in the list of headers. However, it does not preserve any continuation lines which may have been present in the original Content-Type header.

get_content_charset([failobj])
Return the charset parameter of the Content-Type header, coerced to lower case. If there is no Content-Type header, or if that header has no charset parameter, failobj is returned.

Note that this method differs from get_charset() which returns the Charset instance for the default encoding of the message body.

New in version 2.2.2.

get_charsets([failobj])
Return a list containing the character set names in the message. If the message is a multipart, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the charset parameter in the Content-Type header for the represented subpart. However, if the subpart has no Content-Type header, no charset parameter, or is not of the text main MIME type, then that item in the returned list will be failobj.

walk()
: The walk() method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use walk() as the iterator in a for loop; each iteration returns the next subpart.

  Here's an example that prints the MIME type of every part of a multipart message structure:

```
>>> for part in msg.walk():
...     print part.get_content_type()
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
```

Changed in version 2.5: The previously deprecated methods get_type(), get_main_type(), and get_subtype() were removed.

Message objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

preamble
: The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

  The preamble attribute contains this leading extra-armor text for MIME documents. When the Parser discovers some text after the headers but before the first boundary string, it assigns this text to the message's preamble attribute. When the Generator is writing out the plain text representation of a MIME message, and it finds the message has a preamble attribute, it will write this text in the area between the headers and the first boundary. See email.parser and email.generator for details.

  Note that if the message object has no preamble, the preamble attribute will be None.

epilogue
: The epilogue attribute acts the same way as the preamble attribute, except that it contains text that appears between the last boundary and the end of the message.

  Changed in version 2.5: You do not need to set the epilogue to the empty string in order for the Generator to print a newline at the end of the file.

defects
: The defects attribute contains a list of all the problems found when parsing this message. See email.errors for a detailed description of the possible parsing defects.

  New in version 2.4.

18.1.2 email.parser: Parsing email messages

Message object structures can be created in one of two ways: they can be created from whole cloth by instantiating Message objects and stringing them together via attach() and set_payload() calls, or they can be created by parsing a flat text representation of the email message.

The email package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a string or a file object, and the parser will return to you the

root Message instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return True from its is_multipart() method, and the subparts can be accessed via the get_payload() and walk() methods.

There are actually two parser interfaces available for use, the classic Parser API and the incremental FeedParser API. The classic Parser API is fine if you have the entire text of the message in memory as a string, or if the entire message lives in a file on the file system. FeedParser is more appropriate for when you're reading the message from a stream which might block waiting for more input (e.g. reading an email message from a socket). The FeedParser can consume and parse the message incrementally, and only returns the root object when you close the parser[1].

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. There is no magical connection between the email package's bundled parser and the Message class, so your custom parser can create message object trees any way it finds necessary.

FeedParser API

New in version 2.4.

The FeedParser, imported from the email.feedparser module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (e.g. a socket). The FeedParser can of course be used to parse an email message fully contained in a string or a file, but the classic Parser API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The FeedParser's API is simple; you create an instance, feed it a bunch of text until there's no more to feed it, then close the parser to retrieve the root message object. The FeedParser is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object's defects attribute with a list of any problems it found in a message. See the email.errors module for the list of defects that it can find.

Here is the API for the FeedParser:

class email.parser.FeedParser([_factory])

Create a FeedParser instance. Optional _factory is a no-argument callable that will be called whenever a new message object is needed. It defaults to the email.message.Message class.

feed(data)

Feed the FeedParser some more data. data should be a string containing one or more lines. The lines can be partial and the FeedParser will stitch such partial lines together properly. The lines in the string can have any of the common three line endings, carriage return, newline, or carriage return and newline (they can even be mixed).

close()

Closing a FeedParser completes the parsing of all previously fed data, and returns the root message object. It is undefined what happens if you feed more data to a closed FeedParser.

Parser class API

The Parser class, imported from the email.parser module, provides an API that can be used to parse a message when the complete contents of the message are available in a string or file. The email.parser module also provides a second class, called HeaderParser which can be used if you're only interested in the headers

[1] As of email package version 3.0, introduced in Python 2.4, the classic Parser was re-implemented in terms of the FeedParser, so the semantics and results are identical between the two parsers.

of the message. HeaderParser can be much faster in these situations, since it does not attempt to parse the message body, instead setting the payload to the raw body as a string. HeaderParser has the same API as the Parser class.

class email.parser.Parser([_class])
The constructor for the Parser class takes an optional argument _class. This must be a callable factory (such as a function or a class), and it is used whenever a sub-message object needs to be created. It defaults to Message (see email.message). The factory will be called without arguments.

The optional strict flag is ignored.

Deprecated since version 2.4: Because the Parser class is a backward compatible API wrapper around the new-in-Python 2.4 FeedParser, all parsing is effectively non-strict. You should simply stop passing a strict flag to the Parser constructor.

Changed in version 2.2.2: The strict flag was added.

Changed in version 2.4: The strict flag was deprecated.

The other public Parser methods are:

parse(fp[, headersonly])
Read all the data from the file-like object fp, parse the resulting text, and return the root message object. fp must support both the readline() and the read() methods on file-like objects.

The text contained in fp must be formatted as a block of RFC 2822 style headers and header continuation lines, optionally preceded by an envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts).

Optional headersonly is a flag specifying whether to stop parsing after reading the headers or not. The default is False, meaning it parses the entire contents of the file.

Changed in version 2.2.2: The headersonly flag was added.

parsestr(text[, headersonly])
Similar to the parse() method, except it takes a string object instead of a file-like object. Calling this method on a string is exactly equivalent to wrapping text in a StringIO instance first and calling parse().

Optional headersonly is as with the parse() method.

Changed in version 2.2.2: The headersonly flag was added.

Since creating a message object structure from a string or a file object is such a common task, two functions are provided as a convenience. They are available in the top-level email package namespace.

email.message_from_string(s[, _class[, strict]])
Return a message object structure from a string. This is exactly equivalent to Parser().parsestr(s). Optional _class and strict are interpreted as with the Parser class constructor.

Changed in version 2.2.2: The strict flag was added.

email.message_from_file(fp[, _class[, strict]])
Return a message object structure tree from an open file object. This is exactly equivalent to Parser().parse(fp). Optional _class and strict are interpreted as with the Parser class constructor.

Changed in version 2.2.2: The strict flag was added.

Here's an example of how you might use this at an interactive Python prompt:

```
>>> import email
>>> msg = email.message_from_string(myString)
```

Additional notes

Here are some notes on the parsing semantics:

- Most non-multipart type messages are parsed as a single message object with a string payload. These objects will return False for is_multipart(). Their get_payload() method will return a string object.
- All multipart type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return True for is_multipart() and their get_payload() method will return the list of Message subparts.
- Most messages with a content type of message/* (e.g. message/delivery-status and message/rfc822) will also be parsed as container object containing a list payload of length 1. Their is_multipart() method will return True. The single element in the list payload will be a sub-message object.
- Some non-standards compliant messages may not be internally consistent about their multipart-edness. Such messages may have a Content-Type header of type multipart, but their is_multipart() method may return False. If such messages were parsed with the FeedParser, they will have an instance of the MultipartInvariantViolationDefect class in their defects attribute list. See email.errors for details.

18.1.3 email.generator: Generating MIME documents

One of the most common tasks is to generate the flat text of the email message represented by a message object structure. You will need to do this if you want to send your message via the smtplib module or the nntplib module, or print the message on the console. Taking a message object structure and producing a flat text document is the job of the Generator class.

Again, as with the email.parser module, you aren't limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the transformation from flat text, to a message structure via the Parser class, and back to flat text, is idempotent (the input is identical to the output)[1]. On the other hand, using the Generator on a Message constructed by program may result in changes to the Message object as defaults are filled in.

Here are the public methods of the Generator class, imported from the email.generator module:

class email.generator.Generator(outfp[, mangle_from_[, maxheaderlen]])

The constructor for the Generator class takes a file-like object called outfp for an argument. outfp must support the write() method and be usable as the output file in a Python extended print statement.

Optional mangle_from_ is a flag that, when True, puts a > character in front of any line in the body that starts exactly as From, i.e. From followed by a space at the beginning of the line. This is the only guaranteed portable way to avoid having such lines be mistaken for a Unix mailbox format envelope header separator (see WHY THE CONTENT-LENGTH FORMAT IS BAD for details). mangle_from_ defaults to True, but you might want to set this to False if you are not writing Unix mailbox format files.

Optional maxheaderlen specifies the longest length for a non-continued header. When a header line is longer than maxheaderlen (in characters, with tabs expanded to 8 spaces), the header will be split as defined in the Header class. Set to zero to disable header wrapping. The default is 78, as recommended (but not required) by RFC 2822.

The other public Generator methods are:

[1] This statement assumes that you use the appropriate setting for the unixfrom argument, and that you set maxheaderlen=0 (which will preserve whatever the input line lengths were). It is also not strictly true, since in many cases runs of whitespace in headers are collapsed into single blanks. The latter is a bug that will eventually be fixed.

flatten(msg[, unixfrom])
: Print the textual representation of the message object structure rooted at msg to the output file specified when the Generator instance was created. Subparts are visited depth-first and the resulting text will be properly MIME encoded.

  Optional unixfrom is a flag that forces the printing of the envelope header delimiter before the first RFC 2822 header of the root message object. If the root object has no envelope header, a standard one is crafted. By default, this is set to False to inhibit the printing of the envelope delimiter.

  Note that for subparts, no envelope header is ever printed.

  New in version 2.2.2.

clone(fp)
: Return an independent clone of this Generator instance with the exact same options.

  New in version 2.2.2.

write(s)
: Write the string s to the underlying file object, i.e. outfp passed to Generator's constructor. This provides just enough file-like API for Generator instances to be used in extended print statements.

As a convenience, see the methods Message.as_string() and str(aMessage), a.k.a. Message.__str__(), which simplify the generation of a formatted string representation of a message object. For more detail, see email.message.

The email.generator module also provides a derived class, called DecodedGenerator which is like the Generator base class, except that non-text parts are substituted with a format string representing the part.

class email.generator.DecodedGenerator(outfp[, mangle_from_[, maxheaderlen[, fmt]]])
: This class, derived from Generator walks through all the subparts of a message. If the subpart is of main type text, then it prints the decoded payload of the subpart. Optional _mangle_from_ and maxheaderlen are as with the Generator base class.

  If the subpart is not of main type text, optional fmt is a format string that is used instead of the message payload. fmt is expanded with the following keywords, %(keyword)s format:

  - type – Full MIME type of the non-text part
  - maintype – Main MIME type of the non-text part
  - subtype – Sub-MIME type of the non-text part
  - filename – Filename of the non-text part
  - description – Description associated with the non-text part
  - encoding – Content transfer encoding of the non-text part

  The default value for fmt is None, meaning

  ```
  [Non-text (%(type)s) part of message omitted, filename %(filename)s]
  ```

  New in version 2.2.2.

Changed in version 2.5: The previously deprecated method __call__() was removed.

18.1.4 email.mime: Creating email and MIME objects from scratch

Ordinarily, you get a message object structure by passing a file or some text to a parser, which parses the text and returns the root message object. However you can also build a complete message structure from scratch, or even individual Message objects by hand. In fact, you can also take an existing structure and add new Message objects, move them around, etc. This makes a very convenient interface for slicing-and-dicing MIME messages.

You can create a new object structure by creating Message instances, adding attachments and all the appropriate headers manually. For MIME messages though, the email package provides some convenient subclasses to make things easier.

Here are the classes:

class email.mime.base.MIMEBase(_maintype, _subtype, **_params)
Module: email.mime.base

This is the base class for all the MIME-specific subclasses of Message. Ordinarily you won't create instances specifically of MIMEBase, although you could. MIMEBase is provided primarily as a convenient base class for more specific MIME-aware subclasses.

_maintype is the Content-Type major type (e.g. text or image), and _subtype is the Content-Type minor type (e.g. plain or gif). _params is a parameter key/value dictionary and is passed directly to Message.add_header.

The MIMEBase class always adds a Content-Type header (based on _maintype, _subtype, and _params), and a MIME-Version header (always set to 1.0).

class email.mime.nonmultipart.MIMENonMultipart
Module: email.mime.nonmultipart

A subclass of MIMEBase, this is an intermediate base class for MIME messages that are not multipart. The primary purpose of this class is to prevent the use of the attach() method, which only makes sense for multipart messages. If attach() is called, a MultipartConversionError exception is raised.

New in version 2.2.2.

class email.mime.multipart.MIMEMultipart([_subtype[, boundary[, _subparts[, _params]]]])
Module: email.mime.multipart

A subclass of MIMEBase, this is an intermediate base class for MIME messages that are multipart. Optional _subtype defaults to mixed, but can be used to specify the subtype of the message. A Content-Type header of multipart/_subtype will be added to the message object. A MIME-Version header will also be added.

Optional boundary is the multipart boundary string. When None (the default), the boundary is calculated when needed (for example, when the message is serialized).

_subparts is a sequence of initial subparts for the payload. It must be possible to convert this sequence to a list. You can always attach new subparts to the message by using the Message.attach method.

Additional parameters for the Content-Type header are taken from the keyword arguments, or passed into the _params argument, which is a keyword dictionary.

New in version 2.2.2.

class email.mime.application.MIMEApplication(_data[, _subtype[, _encoder[, **_params]]])
Module: email.mime.application

A subclass of MIMENonMultipart, the MIMEApplication class is used to represent MIME message objects of major type application. _data is a string containing the raw byte data. Optional _subtype specifies the MIME subtype and defaults to octet-stream.

Optional _encoder is a callable (i.e. function) which will perform the actual encoding of the data for transport. This callable takes one argument, which is the MIMEApplication instance. It should use get_payload() and set_payload() to change the payload to encoded form. It should also add any Content-Transfer-Encoding or other headers to the message object as necessary. The default encoding is base64. See the email.encoders module for a list of the built-in encoders.

_params are passed straight through to the base class constructor.

New in version 2.5.

class email.mime.audio.MIMEAudio(_audiodata[, _subtype[, _encoder[, **_params]]])
Module: email.mime.audio

A subclass of MIMENonMultipart, the MIMEAudio class is used to create MIME message objects of major type audio. _audiodata is a string containing the raw audio data. If this data can be decoded by the standard Python module sndhdr, then the subtype will be automatically included in the Content-Type header. Otherwise you can explicitly specify the audio subtype via the _subtype parameter. If the minor type could not be guessed and _subtype was not given, then TypeError is raised.

Optional _encoder is a callable (i.e. function) which will perform the actual encoding of the audio data for transport. This callable takes one argument, which is the MIMEAudio instance. It should use get_payload() and set_payload() to change the payload to encoded form. It should also add any Content-Transfer-Encoding or other headers to the message object as necessary. The default encoding is base64. See the email.encoders module for a list of the built-in encoders.

_params are passed straight through to the base class constructor.

class email.mime.image.MIMEImage(_imagedata[, _subtype[, _encoder[, **_params]]])
Module: email.mime.image

A subclass of MIMENonMultipart, the MIMEImage class is used to create MIME message objects of major type image. _imagedata is a string containing the raw image data. If this data can be decoded by the standard Python module imghdr, then the subtype will be automatically included in the Content-Type header. Otherwise you can explicitly specify the image subtype via the _subtype parameter. If the minor type could not be guessed and _subtype was not given, then TypeError is raised.

Optional _encoder is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable takes one argument, which is the MIMEImage instance. It should use get_payload() and set_payload() to change the payload to encoded form. It should also add any Content-Transfer-Encoding or other headers to the message object as necessary. The default encoding is base64. See the email.encoders module for a list of the built-in encoders.

_params are passed straight through to the MIMEBase constructor.

class email.mime.message.MIMEMessage(_msg[, _subtype])
Module: email.mime.message

A subclass of MIMENonMultipart, the MIMEMessage class is used to create MIME objects of main type message. _msg is used as the payload, and must be an instance of class Message (or a subclass thereof), otherwise a TypeError is raised.

Optional _subtype sets the subtype of the message; it defaults to rfc822.

class email.mime.text.MIMEText(_text[, _subtype[, _charset]])
Module: email.mime.text

A subclass of MIMENonMultipart, the MIMEText class is used to create MIME objects of major type text. _text is the string for the payload. _subtype is the minor type and defaults to plain. _charset is the character set of the text and is passed as a parameter to the MIMENonMultipart constructor; it

defaults to us-ascii. If _text is unicode, it is encoded using the output_charset of _charset, otherwise it is used as-is.

Changed in version 2.4: The previously deprecated _encoding argument has been removed. Content Transfer Encoding now happens implicitly based on the _charset argument.

Unless the _charset parameter is explicitly set to None, the MIMEText object created will have both a Content-Type header with a charset parameter, and a Content-Transfer-Encoding header. This means that a subsequent set_payload call will not result in an encoded payload, even if a charset is passed in the set_payload command. You can "reset" this behavior by deleting the Content-Transfer-Encoding header, after which a set_payload call will automatically encode the new payload (and add a new Content-Transfer-Encoding header).

18.1.5 email.header: Internationalized headers

RFC 2822 is the base standard that describes the format of email messages. It derives from the older RFC 822 standard which came into widespread use at a time when most email was composed of ASCII characters only. RFC 2822 is a specification written assuming email contains only 7-bit ASCII characters.

Of course, as email has been deployed worldwide, it has become internationalized, such that language specific character sets can now be used in email messages. The base standard still requires email messages to be transferred using only 7-bit ASCII characters, so a slew of RFCs have been written describing how to encode email containing non-ASCII characters into RFC 2822-compliant format. These RFCs include RFC 2045, RFC 2046, RFC 2047, and RFC 2231. The email package supports these standards in its email.header and email.charset modules.

If you want to include non-ASCII characters in your email headers, say in the Subject or To fields, you should use the Header class and assign the field in the Message object to an instance of Header instead of using a string for the header value. Import the Header class from the email.header module. For example:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> print msg.as_string()
Subject: =?iso-8859-1?q?p=F6stal?=
```

Notice here how we wanted the Subject field to contain a non-ASCII character? We did this by creating a Header instance and passing in the character set that the byte string was encoded in. When the subsequent Message instance was flattened, the Subject field was properly RFC 2047 encoded. MIME-aware mail readers would show this header using the embedded ISO-8859-1 character.

New in version 2.2.2.

Here is the Header class description:

class email.header.Header([s[, charset[, maxlinelen[, header_name[, continuation_ws[, errors]]]]]])
Create a MIME-compliant header that can contain strings in different character sets.

Optional s is the initial header value. If None (the default), the initial header value is not set. You can later append to the header with append() method calls. s may be a byte string or a Unicode string, but see the append() documentation for semantics.

Optional charset serves two purposes: it has the same meaning as the charset argument to the append() method. It also sets the default character set for all subsequent append() calls that omit the charset argument. If charset is not provided in the constructor (the default), the us-ascii character set is used both as s's initial charset and as the default for subsequent append() calls.

The maximum line length can be specified explicitly via maxlinelen. For splitting the first line to a shorter value (to account for the field header which isn't included in s, e.g. Subject) pass in the name of the field in header_name. The default maxlinelen is 76, and the default value for header_name is None, meaning it is not taken into account for the first line of a long, split header.

Optional continuation_ws must be RFC 2822-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines. continuation_ws defaults to a single space character (" ").

Optional errors is passed straight through to the append() method.

append(s[, charset[, errors]])
: Append the string s to the MIME header.

  Optional charset, if given, should be a Charset instance (see email.charset) or the name of a character set, which will be converted to a Charset instance. A value of None (the default) means that the charset given in the constructor is used.

  s may be a byte string or a Unicode string. If it is a byte string (i.e. isinstance(s, str) is true), then charset is the encoding of that byte string, and a UnicodeError will be raised if the string cannot be decoded with that character set.

  If s is a Unicode string, then charset is a hint specifying the character set of the characters in the string. In this case, when producing an RFC 2822-compliant header using RFC 2047 rules, the Unicode string will be encoded using the following charsets in order: us-ascii, the charset hint, utf-8. The first character set to not provoke a UnicodeError is used.

  Optional errors is passed through to any unicode() or unicode.encode() call, and defaults to "strict".

encode([splitchars])
: Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII parts in base64 or quoted-printable encodings. Optional splitchars is a string containing characters to split long ASCII lines on, in rough support of RFC 2822's highest level syntactic breaks. This doesn't affect RFC 2047 encoded lines.

The Header class also provides a number of methods to support standard operators and built-in functions.

__str__()
: A synonym for Header.encode(). Useful for str(aHeader).

__unicode__()
: A helper for the built-in unicode() function. Returns the header as a Unicode string.

__eq__(other)
: This method allows you to compare two Header instances for equality.

__ne__(other)
: This method allows you to compare two Header instances for inequality.

The email.header module also provides the following convenient functions.

email.header.decode_header(header)
: Decode a message header value without converting the character set. The header value is in header.

  This function returns a list of (decoded_string, charset) pairs containing each of the decoded parts of the header. charset is None for non-encoded parts of the header, otherwise a lower case string containing the name of the character set specified in the encoded string.

  Here's an example:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[('p\xf6stal', 'iso-8859-1')]
```

email.header.make_header(decoded_seq[, maxlinelen[, header_name[, continuation_ws]]])
Create a Header instance from a sequence of pairs as returned by decode_header().

decode_header() takes a header value string and returns a sequence of pairs of the format (decoded_string, charset) where charset is the name of the character set.

This function takes one of those sequence of pairs and returns a Header instance. Optional maxlinelen, header_name, and continuation_ws are as in the Header constructor.

18.1.6 email.charset: Representing character sets

This module provides a class Charset for representing character sets and character set conversions in email messages, as well as a character set registry and several convenience methods for manipulating this registry. Instances of Charset are used in several other modules within the email package.

Import this class from the email.charset module.

New in version 2.2.2.

class email.charset.Charset([input_charset])
Map character sets to their email properties.

This class provides information about the requirements imposed on email for a specific character set. It also provides convenience routines for converting between character sets, given the availability of the applicable codecs. Given a character set, it will do its best to provide information on how to use that character set in an email message in an RFC-compliant way.

Certain character sets must be encoded with quoted-printable or base64 when used in email headers or bodies. Certain character sets must be converted outright, and are not allowed in email.

Optional input_charset is as described below; it is always coerced to lower case. After being alias normalized it is also used as a lookup into the registry of character sets to find out the header encoding, body encoding, and output conversion codec to be used for the character set. For example, if input_charset is iso-8859-1, then headers and bodies will be encoded using quoted-printable and no output conversion codec is necessary. If input_charset is euc-jp, then headers will be encoded with base64, bodies will not be encoded, but output text will be converted from the euc-jp character set to the iso-2022-jp character set.

Charset instances have the following data attributes:

input_charset
The initial character set specified. Common aliases are converted to their official email names (e.g. latin_1 is converted to iso-8859-1). Defaults to 7-bit us-ascii.

header_encoding
If the character set must be encoded before it can be used in an email header, this attribute will be set to Charset.QP (for quoted-printable), Charset.BASE64 (for base64 encoding), or Charset.SHORTEST for the shortest of QP or BASE64 encoding. Otherwise, it will be None.

body_encoding
Same as header_encoding, but describes the encoding for the mail message's body, which indeed may be different than the header encoding. Charset.SHORTEST is not allowed for body_encoding.

output_charset
: Some character sets must be converted before they can be used in email headers or bodies. If the input_charset is one of them, this attribute will contain the name of the character set output will be converted to. Otherwise, it will be None.

input_codec
: The name of the Python codec used to convert the input_charset to Unicode. If no conversion codec is necessary, this attribute will be None.

output_codec
: The name of the Python codec used to convert Unicode to the output_charset. If no conversion codec is necessary, this attribute will have the same value as the input_codec.

Charset instances also have the following methods:

get_body_encoding()
: Return the content transfer encoding used for body encoding.

  This is either the string quoted-printable or base64 depending on the encoding used, or it is a function, in which case you should call the function with a single argument, the Message object being encoded. The function should then set the Content-Transfer-Encoding header itself to whatever is appropriate.

  Returns the string quoted-printable if body_encoding is QP, returns the string base64 if body_encoding is BASE64, and returns the string 7bit otherwise.

convert(s)
: Convert the string s from the input_codec to the output_codec.

to_splittable(s)
: Convert a possibly multibyte string to a safely splittable format. s is the string to split.

  Uses the input_codec to try and convert the string to Unicode, so it can be safely split on character boundaries (even for multibyte characters).

  Returns the string as-is if it isn't known how to convert s to Unicode with the input_charset.

  Characters that could not be converted to Unicode will be replaced with the Unicode replacement character `'U+FFFD'`.

from_splittable(ustr[, to_output])
: Convert a splittable string back into an encoded string. ustr is a Unicode string to "unsplit".

  This method uses the proper codec to try and convert the string from Unicode back into an encoded format. Return the string as-is if it is not Unicode, or if it could not be converted from Unicode.

  Characters that could not be converted from Unicode will be replaced with an appropriate character (usually `'?'`).

  If to_output is True (the default), uses output_codec to convert to an encoded format. If to_output is False, it uses input_codec.

get_output_charset()
: Return the output character set.

  This is the output_charset attribute if that is not None, otherwise it is input_charset.

encoded_header_len()
: Return the length of the encoded header string, properly calculating for quoted-printable or base64 encoding.

header_encode(s[, convert])
Header-encode the string s.

If convert is True, the string will be converted from the input charset to the output charset automatically. This is not useful for multibyte character sets, which have line length issues (multibyte characters must be split on a character, not a byte boundary); use the higher-level Header class to deal with these issues (see email.header). convert defaults to False.

The type of encoding (base64 or quoted-printable) will be based on the header_encoding attribute.

body_encode(s[, convert])
Body-encode the string s.

If convert is True (the default), the string will be converted from the input charset to output charset automatically. Unlike header_encode(), there are no issues with byte boundaries and multibyte charsets in email bodies, so this is usually pretty safe.

The type of encoding (base64 or quoted-printable) will be based on the body_encoding attribute.

The Charset class also provides a number of methods to support standard operations and built-in functions.

__str__()
Returns input_charset as a string coerced to lower case. __repr__() is an alias for __str__().

__eq__(other)
This method allows you to compare two Charset instances for equality.

__ne__(other)
This method allows you to compare two Charset instances for inequality.

The email.charset module also provides the following functions for adding new entries to the global character set, alias, and codec registries:

email.charset.add_charset(charset[, header_enc[, body_enc[, output_charset]]])
Add character properties to the global registry.

charset is the input character set, and must be the canonical name of a character set.

Optional header_enc and body_enc is either Charset.QP for quoted-printable, Charset.BASE64 for base64 encoding, Charset.SHORTEST for the shortest of quoted-printable or base64 encoding, or None for no encoding. SHORTEST is only valid for header_enc. The default is None for no encoding.

Optional output_charset is the character set that the output should be in. Conversions will proceed from input charset, to Unicode, to the output charset when the method Charset.convert() is called. The default is to output in the same character set as the input.

Both input_charset and output_charset must have Unicode codec entries in the module's character set-to-codec mapping; use add_codec() to add codecs the module does not know about. See the codecs module's documentation for more information.

The global character set registry is kept in the module global dictionary CHARSETS.

email.charset.add_alias(alias, canonical)
Add a character set alias. alias is the alias name, e.g. latin-1. canonical is the character set's canonical name, e.g. iso-8859-1.

The global charset alias registry is kept in the module global dictionary ALIASES.

email.charset.add_codec(charset, codecname)
Add a codec that map characters in the given character set to and from Unicode.

charset is the canonical name of a character set. codecname is the name of a Python codec, as appropriate for the second argument to the unicode() built-in, or to the encode() method of a Unicode string.

18.1.7 email.encoders: Encoders

When creating Message objects from scratch, you often need to encode the payloads for transport through compliant mail servers. This is especially true for image/* and text/* type messages containing binary data.

The email package provides some convenient encodings in its encoders module. These encoders are actually used by the MIMEAudio and MIMEImage class constructors to provide default encodings. All encoder functions take exactly one argument, the message object to encode. They usually extract the payload, encode it, and reset the payload to this newly encoded value. They should also set the Content-Transfer-Encoding header as appropriate.

Note that these functions are not meaningful for a multipart message. They must be applied to individual subparts instead, and will raise a TypeError if passed a message whose type is multipart.

Here are the encoding functions provided:

email.encoders.encode_quopri(msg)
: Encodes the payload into quoted-printable form and sets the Content-Transfer-Encoding header to quoted-printable[1]. This is a good encoding to use when most of your payload is normal printable data, but contains a few unprintable characters.

email.encoders.encode_base64(msg)
: Encodes the payload into base64 form and sets the Content-Transfer-Encoding header to base64. This is a good encoding to use when most of your payload is unprintable data since it is a more compact form than quoted-printable. The drawback of base64 encoding is that it renders the text non-human readable.

email.encoders.encode_7or8bit(msg)
: This doesn't actually modify the message's payload, but it does set the Content-Transfer-Encoding header to either 7bit or 8bit as appropriate, based on the payload data.

email.encoders.encode_noop(msg)
: This does nothing; it doesn't even set the Content-Transfer-Encoding header.

18.1.8 email.errors: Exception and Defect classes

The following exception classes are defined in the email.errors module:

exception email.errors.MessageError
: This is the base class for all exceptions that the email package can raise. It is derived from the standard Exception class and defines no additional methods.

exception email.errors.MessageParseError
: This is the base class for exceptions raised by the Parser class. It is derived from MessageError.

exception email.errors.HeaderParseError
: Raised under some error conditions when parsing the RFC 2822 headers of a message, this class is derived from MessageParseError. It can be raised from the Parser.parse or Parser.parsestr methods.

  Situations where it can be raised include finding an envelope header after the first RFC 2822 header of the message, finding a continuation line before the first RFC 2822 header is found, or finding a line in the headers which is neither a header or a continuation line.

[1] Note that encoding with encode_quopri() also encodes all tabs and space characters in the data.

exception email.errors.BoundaryError
: Raised under some error conditions when parsing the RFC 2822 headers of a message, this class is derived from MessageParseError. It can be raised from the Parser.parse or Parser.parsestr methods.

  Situations where it can be raised include not being able to find the starting or terminating boundary in a multipart/* message when strict parsing is used.

exception email.errors.MultipartConversionError
: Raised when a payload is added to a Message object using add_payload(), but the payload is already a scalar and the message's Content-Type main type is not either multipart or missing. MultipartConversionError multiply inherits from MessageError and the built-in TypeError.

  Since Message.add_payload() is deprecated, this exception is rarely raised in practice. However the exception may also be raised if the attach() method is called on an instance of a class derived from MIMENonMultipart (e.g. MIMEImage).

Here's the list of the defects that the FeedParser can find while parsing messages. Note that the defects are added to the message where the problem was found, so for example, if a message nested inside a multipart/alternative had a malformed header, that nested message object would have a defect, but the containing messages would not.

All defect classes are subclassed from email.errors.MessageDefect, but this class is not an exception!

New in version 2.4: All the defect classes were added.

- NoBoundaryInMultipartDefect – A message claimed to be a multipart, but had no boundary parameter.
- StartBoundaryNotFoundDefect – The start boundary claimed in the Content-Type header was never found.
- FirstHeaderLineIsContinuationDefect – The message had a continuation line as its first header line.
- MisplacedEnvelopeHeaderDefect - A "Unix From" header was found in the middle of a header block.
- MalformedHeaderDefect – A header was found that was missing a colon, or was otherwise malformed.
- MultipartInvariantViolationDefect – A message claimed to be a multipart, but no subparts were found. Note that when a message has this defect, its is_multipart() method may return false even though its content type claims to be multipart.

18.1.9 email.utils: Miscellaneous utilities

There are several useful utilities provided in the email.utils module:

email.utils.quote(str)
: Return a new string with backslashes in str replaced by two backslashes, and double quotes replaced by backslash-double quote.

email.utils.unquote(str)
: Return a new string which is an unquoted version of str. If str ends and begins with double quotes, they are stripped off. Likewise if str ends and begins with angle brackets, they are stripped off.

email.utils.parseaddr(address)
: Parse address – which should be the value of some address-containing field such as To or Cc – into its constituent realname and email address parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple of `('', '')` is returned.

email.utils.formataddr(pair)
: The inverse of parseaddr(), this takes a 2-tuple of the form (realname, email_address) and returns the string value suitable for a To or Cc header. If the first element of pair is false, then the second element is returned unmodified.

email.utils.getaddresses(fieldvalues)
: This method returns a list of 2-tuples of the form returned by parseaddr(). fieldvalues is a sequence of header field values as might be returned by Message.get_all. Here's a simple example that gets all the recipients of a message:

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

email.utils.parsedate(date)
: Attempts to parse a date according to the rules in RFC 2822. however, some mailers don't follow that format as specified, so parsedate() tries to guess correctly in such cases. date is a string containing an RFC 2822 date, such as "Mon, 20 Nov 1995 19:12:08 -0500". If it succeeds in parsing the date, parsedate() returns a 9-tuple that can be passed directly to time.mktime(); otherwise None will be returned. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.parsedate_tz(date)
: Performs the same function as parsedate(), but returns either None or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to time.mktime(), and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time)[1]. If the input string has no timezone, the last element of the tuple returned is None. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.mktime_tz(tuple)
: Turn a 10-tuple as returned by parsedate_tz() into a UTC timestamp (seconds since the Epoch). If the timezone item in the tuple is None, assume local time.

email.utils.formatdate([timeval[, localtime][, usegmt]])
: Returns a date string as per RFC 2822, e.g.:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

  Optional timeval if given is a floating point time value as accepted by time.gmtime() and time.localtime(), otherwise the current time is used.

  Optional localtime is a flag that when True, interprets timeval, and returns a date relative to the local timezone instead of UTC, properly taking daylight savings time into account. The default is False meaning UTC is used.

  Optional usegmt is a flag that when True, outputs a date string with the timezone as an ascii string GMT, rather than a numeric -0000. This is needed for some protocols (such as HTTP). This only applies when localtime is False. The default is False.

  New in version 2.4.

email.utils.make_msgid([idstring])
: Returns a string suitable for an RFC 2822-compliant Message-ID header. Optional idstring if given, is a string used to strengthen the uniqueness of the message id.

email.utils.decode_rfc2231(s)
: Decode the string s according to RFC 2231.

[1] Note that the sign of the timezone offset is the opposite of the sign of the time.timezone variable for the same timezone; the latter variable follows the POSIX standard while this module follows RFC 2822.

email.utils.encode_rfc2231(s[, charset[, language]])
: Encode the string s according to RFC 2231. Optional charset and language, if given is the character set name and language name to use. If neither is given, s is returned as-is. If charset is given but language is not, the string is encoded using the empty string for language.

email.utils.collapse_rfc2231_value(value[, errors[, fallback_charset]])
: When a header parameter is encoded in RFC 2231 format, Message.get_param may return a 3-tuple containing the character set, language, and value. collapse_rfc2231_value() turns this into a unicode string. Optional errors is passed to the errors argument of the built-in unicode() function; it defaults to replace. Optional fallback_charset specifies the character set to use if the one in the RFC 2231 header is not known by Python; it defaults to us-ascii.

  For convenience, if the value passed to collapse_rfc2231_value() is not a tuple, it should be a string and it is returned unquoted.

email.utils.decode_params(params)
: Decode parameters list according to RFC 2231. params is a sequence of 2-tuples containing elements of the form (content-type, string-value).

Changed in version 2.4: The dump_address_pair() function has been removed; use formataddr() instead.

Changed in version 2.4: The decode() function has been removed; use the Header.decode_header method instead.

Changed in version 2.4: The encode() function has been removed; use the Header.encode method instead.

18.1.10 email.iterators: Iterators

Iterating over a message object tree is fairly easy with the Message.walk method. The email.iterators module provides some useful higher level iterations over message object trees.

email.iterators.body_line_iterator(msg[, decode])
: This iterates over all the payloads in all the subparts of msg, returning the string payloads line-by-line. It skips over all the subpart headers, and it skips over any subpart with a payload that isn't a Python string. This is somewhat equivalent to reading the flat text representation of the message from a file using readline(), skipping over all the intervening headers.

  Optional decode is passed through to Message.get_payload.

email.iterators.typed_subpart_iterator(msg[, maintype[, subtype]])
: This iterates over all the subparts of msg, returning only those subparts that match the MIME type specified by maintype and subtype.

  Note that subtype is optional; if omitted, then subpart MIME type matching is done only with the main type. maintype is optional too; it defaults to text.

  Thus, by default typed_subpart_iterator() returns each subpart that has a MIME type of text/*.

The following function has been added as a useful debugging tool. It should not be considered part of the supported public interface for the package.

email.iterators._structure(msg[, fp[, level]])
: Prints an indented representation of the content types of the message object structure. For example:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
```

(continues on next page)

(continued from previous page)

```
multipart/digest
    message/rfc822
        text/plain
    message/rfc822
        text/plain
    message/rfc822
        text/plain
    message/rfc822
        text/plain
    message/rfc822
        text/plain
text/plain
```

Optional fp is a file-like object to print the output to. It must be suitable for Python's extended print statement. level is used internally.

18.1.11 email: Examples

Here are a few examples of how to use the email package to read, write, and send simple email messages, as well as more complex MIME messages.

First, let's see how to create and send a simple text message:

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.mime.text import MIMEText

# Open a plain text file for reading.  For this example, assume that
# the text file contains only ASCII characters.
fp = open(textfile, 'rb')
# Create a text/plain message
msg = MIMEText(fp.read())
fp.close()

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = 'The contents of %s' % textfile
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server, but don't include the
# envelope header.
s = smtplib.SMTP('localhost')
s.sendmail(me, [you], msg.as_string())
s.quit()
```

And parsing RFC822 headers can easily be done by the parse(filename) or parsestr(message_as_string) methods of the Parser() class:

```
# Import the email modules we'll need
from email.parser import Parser

#  If the e-mail headers are in a file, uncomment this line:
```

(continues on next page)

(continued from previous page)

```
#headers = Parser().parse(open(messagefile, 'r'))

#  Or for parsing headers in a string, use:
headers = Parser().parsestr('From: <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  Now the header items can be accessed as a dictionary:
print 'To: %s' % headers['to']
print 'From: %s' % headers['from']
print 'Subject: %s' % headers['subject']
```

Here's an example of how to send a MIME message containing a bunch of family pictures that may be residing in a directory:

```
# Import smtplib for the actual sending function
import smtplib

# Here are the email package modules we'll need
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart

COMMASPACE = ', '

# Create the container (outer) email message.
msg = MIMEMultipart()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = COMMASPACE.join(family)
msg.preamble = 'Our family reunion'

# Assume we know that the image files are all in PNG format
for file in pngfiles:
    # Open the files in binary mode.  Let the MIMEImage class automatically
    # guess the specific image type.
    fp = open(file, 'rb')
    img = MIMEImage(fp.read())
    fp.close()
    msg.attach(img)

# Send the email via our own SMTP server.
s = smtplib.SMTP('localhost')
s.sendmail(me, family, msg.as_string())
s.quit()
```

Here's an example of how to send the entire contents of a directory as an email message:[1]

```
#!/usr/bin/env python

"""Send the contents of a directory as a MIME message."""
```

(continues on next page)

[1] Thanks to Matthew Dixon Cowles for the original inspiration and examples.

(continued from previous page)

```
import os
import sys
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from optparse import OptionParser

from email import encoders
from email.message import Message
from email.mime.audio import MIMEAudio
from email.mime.base import MIMEBase
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

COMMASPACE = ', '


def main():
    parser = OptionParser(usage="""\
Send the contents of a directory as a MIME message.

Usage: %prog [options]

Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="""Mail the contents of the specified directory,
                      otherwise use the current directory.  Only the regular
                      files in the directory are sent, and we don't recurse to
                      subdirectories.""")
    parser.add_option('-o', '--output',
                      type='string', action='store', metavar='FILE',
                      help="""Print the composed message to FILE instead of
                      sending the message to the SMTP server.""")
    parser.add_option('-s', '--sender',
                      type='string', action='store', metavar='SENDER',
                      help='The value of the From: header (required)')
    parser.add_option('-r', '--recipient',
                      type='string', action='append', metavar='RECIPIENT',
                      default=[], dest='recipients',
                      help='A To: header value (at least one required)')
    opts, args = parser.parse_args()
    if not opts.sender or not opts.recipients:
        parser.print_help()
        sys.exit(1)
    directory = opts.directory
    if not directory:
        directory = '.'
    # Create the enclosing (outer) message
    outer = MIMEMultipart()
    outer['Subject'] = 'Contents of directory %s' % os.path.abspath(directory)
```

(continues on next page)

(continued from previous page)

```
    outer['To'] = COMMASPACE.join(opts.recipients)
    outer['From'] = opts.sender
    outer.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        if maintype == 'text':
            fp = open(path)
            # Note: we should handle calculating the charset
            msg = MIMEText(fp.read(), _subtype=subtype)
            fp.close()
        elif maintype == 'image':
            fp = open(path, 'rb')
            msg = MIMEImage(fp.read(), _subtype=subtype)
            fp.close()
        elif maintype == 'audio':
            fp = open(path, 'rb')
            msg = MIMEAudio(fp.read(), _subtype=subtype)
            fp.close()
        else:
            fp = open(path, 'rb')
            msg = MIMEBase(maintype, subtype)
            msg.set_payload(fp.read())
            fp.close()
            # Encode the payload using Base64
            encoders.encode_base64(msg)
        # Set the filename parameter
        msg.add_header('Content-Disposition', 'attachment', filename=filename)
        outer.attach(msg)
    # Now send or store the message
    composed = outer.as_string()
    if opts.output:
        fp = open(opts.output, 'w')
        fp.write(composed)
        fp.close()
    else:
        s = smtplib.SMTP('localhost')
        s.sendmail(opts.sender, opts.recipients, composed)
        s.quit()


if __name__ == '__main__':
    main()
```

Here's an example of how to unpack a MIME message like the one above, into a directory of files:

```
#!/usr/bin/env python

"""Unpack a MIME message into a directory of files."""

import os
import sys
import email
import errno
import mimetypes

from optparse import OptionParser


def main():
    parser = OptionParser(usage="""\
Unpack a MIME message into a directory of files.

Usage: %prog [options] msgfile
""")
    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="""Unpack the MIME message into the named
                      directory, which will be created if it doesn't already
                      exist.""")
    opts, args = parser.parse_args()
    if not opts.directory:
        parser.print_help()
        sys.exit(1)

    try:
        msgfile = args[0]
    except IndexError:
        parser.print_help()
        sys.exit(1)

    try:
        os.mkdir(opts.directory)
    except OSError as e:
        # Ignore directory exists error
        if e.errno != errno.EEXIST:
            raise

    fp = open(msgfile)
    msg = email.message_from_file(fp)
    fp.close()

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
```

(continues on next page)

(continued from previous page)

```
            # Use a generic bag-of-bits extension
            ext = '.bin'
        filename = 'part-%03d%s' % (counter, ext)
    counter += 1
    fp = open(os.path.join(opts.directory, filename), 'wb')
    fp.write(part.get_payload(decode=True))
    fp.close()


if __name__ == '__main__':
    main()
```

Here's an example of how to create an HTML message with an alternative plain text version:[2]

```
#!/usr/bin/env python

import smtplib

from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

# me == my email address
# you == recipient's email address
me = "my@email.com"
you = "your@email.com"

# Create message container - the correct MIME type is multipart/alternative.
msg = MIMEMultipart('alternative')
msg['Subject'] = "Link"
msg['From'] = me
msg['To'] = you

# Create the body of the message (a plain-text and an HTML version).
text = "Hi!\nHow are you?\nHere is the link you wanted:\nhttps://www.python.org"
html = """\
<html>
  <head></head>
  <body>
    <p>Hi!<br>
       How are you?<br>
       Here is the <a href="https://www.python.org">link</a> you wanted.
    </p>
  </body>
</html>
"""

# Record the MIME types of both parts - text/plain and text/html.
part1 = MIMEText(text, 'plain')
part2 = MIMEText(html, 'html')

# Attach parts into message container.
# According to RFC 2046, the last part of a multipart message, in this case
# the HTML message, is best and preferred.
msg.attach(part1)
msg.attach(part2)
```

(continues on next page)

[2] Contributed by Martin Matejek.

(continued from previous page)

```
# Send the message via local SMTP server.
s = smtplib.SMTP('localhost')
# sendmail function takes 3 arguments: sender's address, recipient's address
# and message to send - here it is sent as one string.
s.sendmail(me, you, msg.as_string())
s.quit()
```

See also:

Module smtplib SMTP protocol client

Module nntplib NNTP protocol client

18.1.12 Package History

This table describes the release history of the email package, corresponding to the version of Python that the package was released with. For purposes of this document, when you see a note about change or added versions, these refer to the Python version the change was made in, not the email package version. This table also describes the Python compatibility of each version of the package.

email version	distributed with	compatible with
1.x	Python 2.2.0 to Python 2.2.1	no longer supported
2.5	Python 2.2.2+ and Python 2.3	Python 2.1 to 2.5
3.0	Python 2.4	Python 2.3 to 2.5
4.0	Python 2.5	Python 2.3 to 2.5

Here are the major differences between email version 4 and version 3:

- All modules have been renamed according to PEP 8 standards. For example, the version 3 module email.Message was renamed to email.message in version 4.
- A new subpackage email.mime was added and all the version 3 email.MIME* modules were renamed and situated into the email.mime subpackage. For example, the version 3 module email.MIMEText was renamed to email.mime.text.

  Note that the version 3 names will continue to work until Python 2.6.
- The email.mime.application module was added, which contains the MIMEApplication class.
- Methods that were deprecated in version 3 have been removed. These include Generator.__call__(), Message.get_type(), Message.get_main_type(), Message.get_subtype().
- Fixes have been added for RFC 2231 support which can change some of the return types for Message.get_param and friends. Under some circumstances, values which used to return a 3-tuple now return simple strings (specifically, if all extended parameter segments were unencoded, there is no language and charset designation expected, so the return type is now a simple string). Also, %-decoding used to be done for both encoded and unencoded segments; this decoding is now done only for encoded segments.

Here are the major differences between email version 3 and version 2:

- The FeedParser class was introduced, and the Parser class was implemented in terms of the FeedParser. All parsing therefore is non-strict, and parsing will make a best effort never to raise an exception. Problems found while parsing messages are stored in the message's defect attribute.

- All aspects of the API which raised DeprecationWarnings in version 2 have been removed. These include the _encoder argument to the MIMEText constructor, the Message.add_payload() method, the Utils.dump_address_pair() function, and the functions Utils.decode() and Utils.encode().
- New DeprecationWarnings have been added to: Generator.__call__(), Message.get_type(), Message.get_main_type(), Message.get_subtype(), and the strict argument to the Parser class. These are expected to be removed in future versions.
- Support for Pythons earlier than 2.3 has been removed.

Here are the differences between email version 2 and version 1:

- The email.Header and email.Charset modules have been added.
- The pickle format for Message instances has changed. Since this was never (and still isn't) formally defined, this isn't considered a backward incompatibility. However if your application pickles and unpickles Message instances, be aware that in email version 2, Message instances now have private variables _charset and _default_type.
- Several methods in the Message class have been deprecated, or their signatures changed. Also, many new methods have been added. See the documentation for the Message class for details. The changes should be completely backward compatible.
- The object structure has changed in the face of message/rfc822 content types. In email version 1, such a type would be represented by a scalar payload, i.e. the container message's is_multipart() returned false, get_payload() was not a list object, but a single Message instance.

  This structure was inconsistent with the rest of the package, so the object representation for message/rfc822 content types was changed. In email version 2, the container does return True from is_multipart(), and get_payload() returns a list containing a single Message item.

  Note that this is one place that backward compatibility could not be completely maintained. However, if you're already testing the return type of get_payload(), you should be fine. You just need to make sure your code doesn't do a set_payload() with a Message instance on a container with a content type of message/rfc822.
- The Parser constructor's strict argument was added, and its parse() and parsestr() methods grew a headersonly argument. The strict flag was also added to functions email.message_from_file() and email.message_from_string().
- Generator.__call__() is deprecated; use Generator.flatten instead. The Generator class has also grown the clone() method.
- The DecodedGenerator class in the email.generator module was added.
- The intermediate base classes MIMENonMultipart and MIMEMultipart have been added, and interposed in the class hierarchy for most of the other MIME-related derived classes.
- The _encoder argument to the MIMEText constructor has been deprecated. Encoding now happens implicitly based on the _charset argument.
- The following functions in the email.Utils module have been deprecated: dump_address_pairs(), decode(), and encode(). The following functions have been added to the module: make_msgid(), decode_rfc2231(), encode_rfc2231(), and decode_params().
- The non-public function email.Iterators._structure() was added.

18.1.13 Differences from mimelib

The email package was originally prototyped as a separate library called mimelib. Changes have been made so that method names are more consistent, and some methods or modules have either been added or removed.

The semantics of some of the methods have also changed. For the most part, any functionality available in mimelib is still available in the email package, albeit often in a different way. Backward compatibility between the mimelib package and the email package was not a priority.

Here is a brief description of the differences between the mimelib and the email packages, along with hints on how to port your applications.

Of course, the most visible difference between the two packages is that the package name has been changed to email. In addition, the top-level package has the following differences:

- messageFromString() has been renamed to message_from_string().
- messageFromFile() has been renamed to message_from_file().

The Message class has the following differences:

- The method asString() was renamed to as_string().
- The method ismultipart() was renamed to is_multipart().
- The get_payload() method has grown a decode optional argument.
- The method getall() was renamed to get_all().
- The method addheader() was renamed to add_header().
- The method gettype() was renamed to get_type().
- The method getmaintype() was renamed to get_main_type().
- The method getsubtype() was renamed to get_subtype().
- The method getparams() was renamed to get_params(). Also, whereas getparams() returned a list of strings, get_params() returns a list of 2-tuples, effectively the key/value pairs of the parameters, split on the '=' sign.
- The method getparam() was renamed to get_param().
- The method getcharsets() was renamed to get_charsets().
- The method getfilename() was renamed to get_filename().
- The method getboundary() was renamed to get_boundary().
- The method setboundary() was renamed to set_boundary().
- The method getdecodedpayload() was removed. To get similar functionality, pass the value 1 to the decode flag of the get_payload() method.
- The method getpayloadastext() was removed. Similar functionality is supported by the DecodedGenerator class in the email.generator module.
- The method getbodyastext() was removed. You can get similar functionality by creating an iterator with typed_subpart_iterator() in the email.iterators module.

The Parser class has no differences in its public interface. It does have some additional smarts to recognize message/delivery-status type messages, which it represents as a Message instance containing separate Message subparts for each header block in the delivery status notification[1].

The Generator class has no differences in its public interface. There is a new class in the email.generator module though, called DecodedGenerator which provides most of the functionality previously available in the Message.getpayloadastext() method.

The following modules and classes have been changed:

[1] Delivery Status Notifications (DSN) are defined in RFC 1894.

- The MIMEBase class constructor arguments _major and _minor have changed to _maintype and _subtype respectively.
- The Image class/module has been renamed to MIMEImage. The _minor argument has been renamed to _subtype.
- The Text class/module has been renamed to MIMEText. The _minor argument has been renamed to _subtype.
- The MessageRFC822 class/module has been renamed to MIMEMessage. Note that an earlier version of mimelib called this class/module RFC822, but that clashed with the Python standard library module rfc822 on some case-insensitive file systems.

  Also, the MIMEMessage class now represents any kind of MIME message with main type message. It takes an optional argument _subtype which is used to set the MIME subtype. _subtype defaults to rfc822.

mimelib provided some utility functions in its address and date modules. All of these functions have been moved to the email.utils module.

The MsgReader class/module has been removed. Its functionality is most closely supported in the body_line_iterator() function in the email.iterators module.

18.2 json — JSON encoder and decoder

New in version 2.6.

JSON (JavaScript Object Notation), specified by RFC 7159 (which obsoletes RFC 4627) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript[1]).

json exposes an API familiar to users of the standard library marshal and pickle modules.

Encoding basic Python object hierarchies:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print json.dumps("\"foo\bar")
"\"foo\bar"
>>> print json.dumps(u'\u1234')
"\u1234"
>>> print json.dumps('\\')
"\\"
>>> print json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True)
{"a": 0, "b": 0, "c": 0}
>>> from StringIO import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

Compact encoding:

[1] As noted in the errata for RFC 7159, JSON permits literal U+2028 (LINE SEPARATOR) and U+2029 (PARAGRAPH SEPARATOR) characters in strings, whereas JavaScript (as of ECMAScript Edition 5.1) does not.

```
>>> import json
>>> json.dumps([1,2,3,{'4': 5, '6': 7}], separators=(',',':'))
'[1,2,3,{"4":5,"6":7}]'
```

Pretty printing:

```
>>> import json
>>> print json.dumps({'4': 5, '6': 7}, sort_keys=True,
...                  indent=4, separators=(',', ': '))
{
    "4": 5,
    "6": 7
}
```

Decoding JSON:

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
[u'foo', {u'bar': [u'baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
u'"foo\x08ar'
>>> from StringIO import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
[u'streaming API']
```

Specializing JSON object decoding:

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

Extending JSONEncoder:

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[', '2.0', ', ', '1.0', ']']
```

Using json.tool from the shell to validate and pretty-print:

```
$ echo '{"json":"obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -mjson.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

Note: JSON is a subset of YAML 1.2. The JSON produced by this module's default settings (in particular, the default separators value) is also a subset of YAML 1.0 and 1.1. This module can thus also be used as a YAML serializer.

18.2.1 Basic Usage

json.dump(obj, fp, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, encoding="utf-8", default=None, sort_keys=False, **kw)

Serialize obj as a JSON formatted stream to fp (a .write()-supporting file-like object) using this conversion table.

If skipkeys is true (default: False), then dict keys that are not of a basic type (str, unicode, int, long, float, bool, None) will be skipped instead of raising a TypeError.

If ensure_ascii is true (the default), all non-ASCII characters in the output are escaped with \uXXXX sequences, and the result is a str instance consisting of ASCII characters only. If ensure_ascii is false, some chunks written to fp may be unicode instances. This usually happens because the input contains unicode strings or the encoding parameter is used. Unless fp.write() explicitly understands unicode (as in codecs.getwriter()) this is likely to cause an error.

If check_circular is false (default: True), then the circular reference check for container types will be skipped and a circular reference will result in an OverflowError (or worse).

If allow_nan is false (default: True), then it will be a ValueError to serialize out of range float values (nan, inf, -inf) in strict compliance of the JSON specification. If allow_nan is true, their JavaScript equivalents (NaN, Infinity, -Infinity) will be used.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, or negative, will only insert newlines. None (the default) selects the most compact representation.

Note: Since the default item separator is ', ', the output might include trailing whitespace when indent is specified. You can use separators=(',', ': ') to avoid this.

If specified, separators should be an (item_separator, key_separator) tuple. By default, (', ', ': ') are used. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

encoding is the character encoding for str instances, default is UTF-8.

If specified, default should be a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a TypeError. If not specified, TypeError is raised.

If sort_keys is true (default: False), then the output of dictionaries will be sorted by key.

To use a custom JSONEncoder subclass (e.g. one that overrides the default() method to serialize additional types), specify it with the cls kwarg; otherwise JSONEncoder is used.

Note: Unlike pickle and marshal, JSON is not a framed protocol so trying to serialize more objects with repeated calls to dump() and the same fp will result in an invalid JSON file.

---

json.dumps(obj, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, encoding="utf-8", default=None, sort_keys=False, **kw)

Serialize obj to a JSON formatted str using this conversion table. If ensure_ascii is false, the result may contain non-ASCII characters and the return value may be a unicode instance.

The arguments have the same meaning as in dump().

Note: Keys in key/value pairs of JSON are always of the type str. When a dictionary is converted into JSON, all the keys of the dictionary are coerced to strings. As a result of this, if a dictionary is converted into JSON and then back into a dictionary, the dictionary may not equal the original one. That is, loads(dumps(x)) != x if x has non-string keys.

---

json.load(fp[, encoding[, cls[, object_hook[, parse_float[, parse_int[, parse_constant[, object_pairs_hook[, **kw]]]]]]]])

Deserialize fp (a .read()-supporting file-like object containing a JSON document) to a Python object using this conversion table.

If the contents of fp are encoded with an ASCII based encoding other than UTF-8 (e.g. latin-1), then an appropriate encoding name must be specified. Encodings that are not ASCII based (such as UCS-2) are not allowed, and should be wrapped with codecs.getreader(encoding)(fp), or simply decoded to a unicode object and passed to loads().

object_hook is an optional function that will be called with the result of any object literal decoded (a dict). The return value of object_hook will be used instead of the dict. This feature can be used to implement custom decoders (e.g. JSON-RPC class hinting).

object_pairs_hook is an optional function that will be called with the result of any object literal decoded with an ordered list of pairs. The return value of object_pairs_hook will be used instead of the dict. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, collections.OrderedDict() will remember the order of insertion). If object_hook is also defined, the object_pairs_hook takes priority.

Changed in version 2.7: Added support for object_pairs_hook.

parse_float, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to float(num_str). This can be used to use another datatype or parser for JSON floats (e.g. decimal.Decimal).

parse_int, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to int(num_str). This can be used to use another datatype or parser for JSON integers (e.g. float).

parse_constant, if specified, will be called with one of the following strings: '-Infinity', 'Infinity', 'NaN'. This can be used to raise an exception if invalid JSON numbers are encountered.

Changed in version 2.7: parse_constant doesn't get called on 'null', 'true', 'false' anymore.

To use a custom JSONDecoder subclass, specify it with the cls kwarg; otherwise JSONDecoder is used. Additional keyword arguments will be passed to the constructor of the class.

json.loads(s[, encoding[, cls[, object_hook[, parse_float[, parse_int[, parse_constant[, object_pairs_hook[, **kw]]]]]]]])

Deserialize s (a str or unicode instance containing a JSON document) to a Python object using this conversion table.

If s is a str instance and is encoded with an ASCII based encoding other than UTF-8 (e.g. latin-1), then an appropriate encoding name must be specified. Encodings that are not ASCII based (such as UCS-2) are not allowed and should be decoded to unicode first.

The other arguments have the same meaning as in load().

18.2.2 Encoders and Decoders

class json.JSONDecoder([encoding[, object_hook[, parse_float[, parse_int[, parse_constant[, strict[, object_pairs_hook]]]]]]])

Simple JSON decoder.

Performs the following translations in decoding by default:

JSON	Python
object	dict
array	list
string	unicode
number (int)	int, long
number (real)	float
true	True
false	False
null	None

It also understands NaN, Infinity, and -Infinity as their corresponding float values, which is outside the JSON spec.

encoding determines the encoding used to interpret any str objects decoded by this instance (UTF-8 by default). It has no effect when decoding unicode objects.

Note that currently only encodings that are a superset of ASCII work, strings of other encodings should be passed in as unicode.

object_hook, if specified, will be called with the result of every JSON object decoded and its return value will be used in place of the given dict. This can be used to provide custom deserializations (e.g. to support JSON-RPC class hinting).

object_pairs_hook, if specified will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of object_pairs_hook will be used instead of the dict. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, collections.OrderedDict() will remember the order of insertion). If object_hook is also defined, the object_pairs_hook takes priority.

Changed in version 2.7: Added support for object_pairs_hook.

parse_float, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to float(num_str). This can be used to use another datatype or parser for JSON floats (e.g. decimal.Decimal).

parse_int, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to int(num_str). This can be used to use another datatype or parser for JSON integers (e.g. float).

parse_constant, if specified, will be called with one of the following strings: '-Infinity', 'Infinity', 'NaN'. This can be used to raise an exception if invalid JSON numbers are encountered.

If strict is false (True is the default), then control characters will be allowed inside strings. Control characters in this context are those with character codes in the 0–31 range, including '\t' (tab), '\n', '\r' and '\0'.

If the data being deserialized is not a valid JSON document, a ValueError will be raised.

decode(s)
Return the Python representation of s (a str or unicode instance containing a JSON document).

raw_decode(s)
Decode a JSON document from s (a str or unicode beginning with a JSON document) and return a 2-tuple of the Python representation and the index in s where the document ended.

This can be used to decode a JSON document from a string that may have extraneous data at the end.

class json.JSONEncoder([skipkeys[, ensure_ascii[, check_circular[, allow_nan[, sort_keys[, indent[, separators[, encoding[, default]]]]]]]]])
Extensible JSON encoder for Python data structures.

Supports the following objects and types by default:

Python	JSON
dict	object
list, tuple	array
str, unicode	string
int, long, float	number
True	true
False	false
None	null

To extend this to recognize other objects, subclass and implement a default() method with another method that returns a serializable object for o if possible, otherwise it should call the superclass implementation (to raise TypeError).

If skipkeys is false (the default), then it is a TypeError to attempt encoding of keys that are not str, int, long, float or None. If skipkeys is true, such items are simply skipped.

If ensure_ascii is true (the default), all non-ASCII characters in the output are escaped with \uXXXX sequences, and the results are str instances consisting of ASCII characters only. If ensure_ascii is false, a result may be a unicode instance. This usually happens if the input contains unicode strings or the encoding parameter is used.

If check_circular is true (the default), then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true (the default), then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true (default: False), then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer (it is None by default), then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

Note: Since the default item separator is ', ', the output might include trailing whitespace when indent is specified. You can use separators=(',', ': ') to avoid this.

If specified, separators should be an (item_separator, key_separator) tuple. By default, (', ', ': ') are used. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default should be a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a TypeError. If not specified, TypeError is raised.

If encoding is not None, then all input strings will be transformed into unicode using that encoding prior to JSON-encoding. The default is UTF-8.

default(o)
: Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a TypeError).

  For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

encode(o)
: Return a JSON string representation of a Python data structure, o. For example:

```
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(o)
: Encode the given object, o, and yield each string representation as available. For example:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

18.2.3 Standard Compliance and Interoperability

The JSON format is specified by RFC 7159 and by ECMA-404. This section details this module's level of compliance with the RFC. For simplicity, JSONEncoder and JSONDecoder subclasses, and parameters other than those explicitly mentioned, are not considered.

This module does not comply with the RFC in a strict fashion, implementing some extensions that are valid JavaScript but not valid JSON. In particular:

- Infinite and NaN number values are accepted and output;
- Repeated names within an object are accepted, and only the value of the last name-value pair is used.

Since the RFC permits RFC-compliant parsers to accept input texts that are not RFC-compliant, this module's deserializer is technically RFC-compliant under default settings.

Character Encodings

The RFC requires that JSON be represented using either UTF-8, UTF-16, or UTF-32, with UTF-8 being the recommended default for maximum interoperability. Accordingly, this module uses UTF-8 as the default for its encoding parameter.

This module's deserializer only directly works with ASCII-compatible encodings; UTF-16, UTF-32, and other ASCII-incompatible encodings require the use of workarounds described in the documentation for the deserializer's encoding parameter.

As permitted, though not required, by the RFC, this module's serializer sets ensure_ascii=True by default, thus escaping the output so that the resulting strings only contain ASCII characters.

The RFC prohibits adding a byte order mark (BOM) to the start of a JSON text, and this module's serializer does not add a BOM to its output. The RFC permits, but does not require, JSON deserializers to ignore an initial BOM in their input. This module's deserializer raises a ValueError when an initial BOM is present.

The RFC does not explicitly forbid JSON strings which contain byte sequences that don't correspond to valid Unicode characters (e.g. unpaired UTF-16 surrogates), but it does note that they may cause interoperability problems. By default, this module accepts and outputs (when present in the original str) code points for such sequences.

Infinite and NaN Number Values

The RFC does not permit the representation of infinite or NaN number values. Despite that, by default, this module accepts and outputs Infinity, -Infinity, and NaN as if they were valid JSON number literal values:

```
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

In the serializer, the allow_nan parameter can be used to alter this behavior. In the deserializer, the parse_constant parameter can be used to alter this behavior.

Repeated Names Within an Object

The RFC specifies that the names within a JSON object should be unique, but does not mandate how repeated names in JSON objects should be handled. By default, this module does not raise an exception; instead, it ignores all but the last name-value pair for a given name:

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{u'x': 3}
```

The object_pairs_hook parameter can be used to alter this behavior.

Top-level Non-Object, Non-Array Values

The old version of JSON specified by the obsolete RFC 4627 required that the top-level value of a JSON text must be either a JSON object or array (Python dict or list), and could not be a JSON null, boolean, number, or string value. RFC 7159 removed that restriction, and this module does not and has never implemented that restriction in either its serializer or its deserializer.

Regardless, for maximum interoperability, you may wish to voluntarily adhere to the restriction yourself.

Implementation Limitations

Some JSON deserializer implementations may set limits on:

- the size of accepted JSON texts
- the maximum level of nesting of JSON objects and arrays
- the range and precision of JSON numbers
- the content and maximum length of JSON strings

This module does not impose any such limits beyond those of the relevant Python datatypes themselves or the Python interpreter itself.

When serializing to JSON, beware any such limitations in applications that may consume your JSON. In particular, it is common for JSON numbers to be deserialized into IEEE 754 double precision numbers and thus subject to that representation's range and precision limitations. This is especially relevant when serializing Python int values of extremely large magnitude, or when serializing instances of "exotic" numerical types such as decimal.Decimal.

18.3 mailcap — Mailcap file handling

Source code: Lib/mailcap.py

Mailcap files are used to configure how MIME-aware applications such as mail readers and Web browsers react to files with different MIME types. (The name "mailcap" is derived from the phrase "mail capability".) For example, a mailcap file might contain a line like video/mpeg; xmpeg %s. Then, if the user encounters an email message or Web document with the MIME type video/mpeg, %s will be replaced by a filename (usually one belonging to a temporary file) and the xmpeg program can be automatically started to view the file.

The mailcap format is documented in RFC 1524, "A User Agent Configuration Mechanism For Multimedia Mail Format Information," but is not an Internet standard. However, mailcap files are supported on most Unix systems.

mailcap.findmatch(caps, MIMEtype[, key[, filename[, plist]]])

Return a 2-tuple; the first element is a string containing the command line to be executed (which can be passed to os.system()), and the second element is the mailcap entry for a given MIME type. If no matching MIME type can be found, (None, None) is returned.

key is the name of the field desired, which represents the type of activity to be performed; the default value is 'view', since in the most common case you simply want to view the body of the MIME-typed data. Other possible values might be 'compose' and 'edit', if you wanted to create a new body of the given MIME type or alter the existing body data. See RFC 1524 for a complete list of these fields.

filename is the filename to be substituted for %s in the command line; the default value is '/dev/null' which is almost certainly not what you want, so usually you'll override it by specifying a filename.

plist can be a list containing named parameters; the default value is simply an empty list. Each entry in the list must be a string containing the parameter name, an equals sign ('='), and the parameter's value. Mailcap entries can contain named parameters like %{foo}, which will be replaced by the value of the parameter named 'foo'. For example, if the command line showpartial %{id} %{number} %{total} was in a mailcap file, and plist was set to ['id=1', 'number=2', 'total=3'], the resulting command line would be 'showpartial 1 2 3'.

In a mailcap file, the "test" field can optionally be specified to test some external condition (such as the machine architecture, or the window system in use) to determine whether or not the mailcap line applies. findmatch() will automatically check such conditions and skip the entry if the check fails.

mailcap.getcaps()
Returns a dictionary mapping MIME types to a list of mailcap file entries. This dictionary must be passed to the findmatch() function. An entry is stored as a list of dictionaries, but it shouldn't be necessary to know the details of this representation.

The information is derived from all of the mailcap files found on the system. Settings in the user's mailcap file $HOME/.mailcap will override settings in the system mailcap files /etc/mailcap, /usr/etc/mailcap, and /usr/local/etc/mailcap.

An example usage:

```
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

18.4 mailbox — Manipulate mailboxes in various formats

This module defines two classes, Mailbox and Message, for accessing and manipulating on-disk mailboxes and the messages they contain. Mailbox offers a dictionary-like mapping from keys to messages. Message extends the email.message module's Message class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

See also:

Module email Represent and manipulate messages.

18.4.1 Mailbox objects

class mailbox.Mailbox
A mailbox, which may be inspected and modified.

The Mailbox class defines an interface and is not intended to be instantiated. Instead, format-specific subclasses should inherit from Mailbox and your code should instantiate a particular subclass.

The Mailbox interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the Mailbox instance with which they will be used and are only meaningful to that Mailbox instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message.

Messages may be added to a Mailbox instance using the set-like method add() and removed using a del statement or the set-like methods remove() and discard().

Mailbox interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a Message instance) is generated based upon the current state of the mailbox. Similarly, when a message is added to a Mailbox instance, the provided message representation's contents are copied. In neither case is a reference to the message representation kept by the Mailbox instance.

The default Mailbox iterator iterates over message representations, not keys as the default dictionary iterator does. Moreover, modification of a mailbox during iteration is safe and well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator yields them will be silently skipped, though using a key from an iterator may result in a KeyError exception if the corresponding message is subsequently removed.

> Warning: Be very cautious when modifying mailboxes that might be simultaneously changed by some other process. The safest mailbox format to use for such tasks is Maildir; try to avoid using single-file formats such as mbox for concurrent writing. If you're modifying a mailbox, you must lock it by calling the lock() and unlock() methods before reading any messages in the file or making any changes by adding or deleting a message. Failing to lock the mailbox runs the risk of losing messages or corrupting the entire mailbox.

Mailbox instances have the following methods:

add(message)
: Add message to the mailbox and return the key that has been assigned to it.

  Parameter message may be a Message instance, an email.message.Message instance, a string, or a file-like object (which should be open in text mode). If message is an instance of the appropriate format-specific Message subclass (e.g., if it's an mboxMessage instance and this is an mbox instance), its format-specific information is used. Otherwise, reasonable defaults for format-specific information are used.

remove(key)
__delitem__(key)
discard(key)
: Delete the message corresponding to key from the mailbox.

  If no such message exists, a KeyError exception is raised if the method was called as remove() or __delitem__() but no exception is raised if the method was called as discard(). The behavior of discard() may be preferred if the underlying mailbox format supports concurrent modification by other processes.

__setitem__(key, message)
: Replace the message corresponding to key with message. Raise a KeyError exception if no message already corresponds to key.

  As with add(), parameter message may be a Message instance, an email.message.Message instance, a string, or a file-like object (which should be open in text mode). If message is an instance of the appropriate format-specific Message subclass (e.g., if it's an mboxMessage instance and this is an mbox instance), its format-specific information is used. Otherwise, the format-specific information of the message that currently corresponds to key is left unchanged.

iterkeys()
keys()
: Return an iterator over all keys if called as iterkeys() or return a list of keys if called as keys().

itervalues()
__iter__()

values()
: Return an iterator over representations of all messages if called as itervalues() or __iter__() or return a list of such representations if called as values(). The messages are represented as instances of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

  Note: The behavior of __iter__() is unlike that of dictionaries, which iterate over keys.

iteritems()
items()
: Return an iterator over (key, message) pairs, where key is a key and message is a message representation, if called as iteritems() or return a list of such pairs if called as items(). The messages are represented as instances of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

get(key, default=None)
__getitem__(key)
: Return a representation of the message corresponding to key. If no such message exists, default is returned if the method was called as get() and a KeyError exception is raised if the method was called as __getitem__(). The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

get_message(key)
: Return a representation of the message corresponding to key as an instance of the appropriate format-specific Message subclass, or raise a KeyError exception if no such message exists.

get_string(key)
: Return a string representation of the message corresponding to key, or raise a KeyError exception if no such message exists.

get_file(key)
: Return a file-like representation of the message corresponding to key, or raise a KeyError exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

  Note: Unlike other representations of messages, file-like representations are not necessarily independent of the Mailbox instance that created them or of the underlying mailbox. More specific documentation is provided by each subclass.

has_key(key)
__contains__(key)
: Return True if key corresponds to a message, False otherwise.

__len__()
: Return a count of messages in the mailbox.

clear()
: Delete all messages from the mailbox.

pop(key[, default])
: Return a representation of the message corresponding to key and delete the message. If no such message exists, return default if it was supplied or else raise a KeyError exception. The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

popitem()
: Return an arbitrary (key, message) pair, where key is a key and message is a message representation, and delete the corresponding message. If the mailbox is empty, raise a KeyError exception. The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

update(arg)
: Parameter arg should be a key-to-message mapping or an iterable of (key, message) pairs. Updates the mailbox so that, for each given key and message, the message corresponding to key is set to message as if by using __setitem__(). As with __setitem__(), each key must already correspond to a message in the mailbox or else a KeyError exception will be raised, so in general it is incorrect for arg to be a Mailbox instance.

Note: Unlike with dictionaries, keyword arguments are not supported.

---

flush()
: Write any pending changes to the filesystem. For some Mailbox subclasses, changes are always written immediately and flush() does nothing, but you should still make a habit of calling this method.

lock()
: Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An ExternalClashError is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should always lock the mailbox before making any modifications to its contents.

unlock()
: Release the lock on the mailbox, if any.

close()
: Flush the mailbox, unlock it if necessary, and close any open files. For some Mailbox subclasses, this method does nothing.

Maildir

class mailbox.Maildir(dirname, factory=rfc822.Message, create=True)
: A subclass of Mailbox for mailboxes in Maildir format. Parameter factory is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, MaildirMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

  It is for historical reasons that factory defaults to rfc822.Message and that dirname is named as such rather than path. For a Maildir instance that behaves like instances of other Mailbox subclasses, set factory to None.

  Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

  Maildir mailboxes contain three subdirectories, namely: tmp, new, and cur. Messages are created momentarily in the tmp subdirectory and then moved to the new subdirectory to finalize delivery. A mail user agent may subsequently move the message to the cur subdirectory and store information about the state of the message in a special "info" section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if `'.'` is the first character in its name. Folder names are represented by Maildir without the leading `'.'`. Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using `'.'` to delimit levels, e.g., "Archived.2005.07".

Note: The Maildir specification requires the use of a colon (`':'`) in certain message file names. However, some operating systems do not permit this character in file names, If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point (`'!'`) is a popular choice. For example:

```
import mailbox
mailbox.Maildir.colon = '!'
```

The colon attribute may also be set on a per-instance basis.

Maildir instances have all of the methods of Mailbox in addition to the following:

list_folders()
: Return a list of the names of all folders.

get_folder(folder)
: Return a Maildir instance representing the folder whose name is folder. A NoSuchMailboxError exception is raised if the folder does not exist.

add_folder(folder)
: Create a folder whose name is folder and return a Maildir instance representing it.

remove_folder(folder)
: Delete the folder whose name is folder. If the folder contains any messages, a NotEmptyError exception will be raised and the folder will not be deleted.

clean()
: Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

Some Mailbox methods implemented by Maildir deserve special remarks:

add(message)
__setitem__(key, message)
update(arg)

> Warning: These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

flush()
: All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()
unlock()
: Maildir mailboxes do not support (or require) locking, so these methods do nothing.

close()
: Maildir instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

get_file(key)
: Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

See also:

maildir man page from qmail The original specification of the format.

Using maildir format Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on "info" semantics.

maildir man page from Courier Another specification of the format. Describes a common extension for supporting folders.

mbox

class mailbox.mbox(path, factory=None, create=True)
: A subclass of Mailbox for mailboxes in mbox format. Parameter factory is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, mboxMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

  The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are "From ".

  Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, mbox implements the original format, which is sometimes referred to as mboxo. This means that the Content-Length header, if present, is ignored and that any occurrences of "From " at the beginning of a line in a message body are transformed to ">From " when storing the message, although occurrences of ">From " are not transformed to "From " when reading the message.

  Some Mailbox methods implemented by mbox deserve special remarks:

  get_file(key)
  : Using the file after calling flush() or close() on the mbox instance may yield unpredictable results or raise an exception.

  lock()
  unlock()
  : Three locking mechanisms are used—dot locking and, if available, the flock() and lockf() system calls.

See also:

mbox man page from qmail A specification of the format and its variations.

mbox man page from tin Another specification of the format, with details on locking.

Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad An argument for using the original mbox format rather than a variation.

"mbox" is a family of several mutually incompatible mailbox formats A history of mbox variations.

MH

class mailbox.MH(path, factory=None, create=True)

A subclass of Mailbox for mailboxes in MH format. Parameter factory is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, MHMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called folders) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support sequences, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called .mh_sequences in each folder.

The MH class manipulates MH mailboxes, but it does not attempt to emulate all of mh's behaviors. In particular, it does not modify and is not affected by the context or .mh_profile files that are used by mh to store its state and configuration.

MH instances have all of the methods of Mailbox in addition to the following:

list_folders()
: Return a list of the names of all folders.

get_folder(folder)
: Return an MH instance representing the folder whose name is folder. A NoSuchMailboxError exception is raised if the folder does not exist.

add_folder(folder)
: Create a folder whose name is folder and return an MH instance representing it.

remove_folder(folder)
: Delete the folder whose name is folder. If the folder contains any messages, a NotEmptyError exception will be raised and the folder will not be deleted.

get_sequences()
: Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

set_sequences(sequences)
: Re-define the sequences that exist in the mailbox based upon sequences, a dictionary of names mapped to key lists, like returned by get_sequences().

pack()
: Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly.

Note: Already-issued keys are invalidated by this operation and should not be subsequently used.

Some Mailbox methods implemented by MH deserve special remarks:

remove(key)
__delitem__(key)
discard(key)
: These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

lock()
unlock()
: Three locking mechanisms are used—dot locking and, if available, the flock() and lockf() system

calls. For MH mailboxes, locking the mailbox means locking the .mh_sequences file and, only for the duration of any operations that affect them, locking individual message files.

get_file(key)
Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

flush()
All changes to MH mailboxes are immediately applied, so this method does nothing.

close()
MH instances do not keep any open files, so this method is equivalent to unlock().

See also:

nmh - Message Handling System Home page of nmh, an updated version of the original mh.

MH & nmh: Email for Users & Programmers A GPL-licensed book on mh and nmh, with some information on the mailbox format.

Babyl

class mailbox.Babyl(path, factory=None, create=True)
A subclass of Mailbox for mailboxes in Babyl format. Parameter factory is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, BabylMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore (`'\037'`) and Control-L (`'\014'`). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore (`'\037'`) character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically a subset of the original headers that have been reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of labels, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

Babyl instances have all of the methods of Mailbox in addition to the following:

get_labels()
Return a list of the names of all user-defined labels used in the mailbox.

Note: The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some Mailbox methods implemented by Babyl deserve special remarks:

get_file(key)
In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into a StringIO instance (from the StringIO module), which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

lock()

unlock()
: Three locking mechanisms are used—dot locking and, if available, the flock() and lockf() system calls.

See also:

Format of Version 5 Babyl Files A specification of the Babyl format.

Reading Mail with Rmail The Rmail manual, with some information on Babyl semantics.

MMDF

class mailbox.MMDF(path, factory=None, create=True)
: A subclass of Mailbox for mailboxes in MMDF format. Parameter factory is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, MMDFMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

  MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A (`'\001'`) characters. As with the mbox format, the beginning of each message is indicated by a line whose first five characters are "From ", but additional occurrences of "From " are not transformed to ">From " when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.

  Some Mailbox methods implemented by MMDF deserve special remarks:

  get_file(key)
  : Using the file after calling flush() or close() on the MMDF instance may yield unpredictable results or raise an exception.

  lock()
  unlock()
  : Three locking mechanisms are used—dot locking and, if available, the flock() and lockf() system calls.

See also:

mmdf man page from tin A specification of MMDF format from the documentation of tin, a newsreader.

MMDF A Wikipedia article describing the Multichannel Memorandum Distribution Facility.

18.4.2 Message objects

class mailbox.Message([message])
: A subclass of the email.message module's Message. Subclasses of mailbox.Message add mailbox-format-specific state and behavior.

  If message is omitted, the new instance is created in a default, empty state. If message is an email.message.Message instance, its contents are copied; furthermore, any format-specific information is converted insofar as possible if message is a Message instance. If message is a string or a file, it should contain an RFC 2822-compliant message, which is read and parsed.

  The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable

to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that Message instances be used to represent messages retrieved using Mailbox instances. In some situations, the time and memory required to generate Message representations might not be acceptable. For such situations, Mailbox instances also offer string and file-like representations, and a custom message factory may be specified when a Mailbox instance is initialized.

MaildirMessage

class mailbox.MaildirMessage([message])

A message with Maildir-specific behaviors. Parameter message has the same meaning as with the Message constructor.

Typically, a mail user agent application moves all of the messages in the new subdirectory to the cur subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they've actually been read. Each message in cur has an "info" section added to its file name to store information about its state. (Some mail readers may also add an "info" section to messages in new.) The "info" section may take one of two forms: it may contain "2," followed by a list of standardized flags (e.g., "2,FR") or it may contain "1," followed by so-called experimental information. Standard flags for Maildir messages are as follows:

Flag	Meaning	Explanation
D	Draft	Under composition
F	Flagged	Marked as important
P	Passed	Forwarded, resent, or bounced
R	Replied	Replied to
S	Seen	Read
T	Trashed	Marked for subsequent deletion

MaildirMessage instances offer the following methods:

get_subdir()

Return either "new" (if the message should be stored in the new subdirectory) or "cur" (if the message should be stored in the cur subdirectory).

Note: A message is typically moved from new to cur after its mailbox has been accessed, whether or not the message is has been read. A message msg has been read if "S" in msg.get_flags() is True.

---

set_subdir(subdir)

Set the subdirectory the message should be stored in. Parameter subdir must be either "new" or "cur".

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of 'D', 'F', 'P', 'R', 'S', and 'T'. The empty string is returned if no flags are set or if "info" contains experimental semantics.

set_flags(flags)

Set the flags specified by flags and unset all others.

add_flag(flag)
: Set the flag(s) specified by flag without changing other flags. To add more than one flag at a time, flag may be a string of more than one character. The current "info" is overwritten whether or not it contains experimental information rather than flags.

remove_flag(flag)
: Unset the flag(s) specified by flag without changing other flags. To remove more than one flag at a time, flag maybe a string of more than one character. If "info" contains experimental information rather than flags, the current "info" is not modified.

get_date()
: Return the delivery date of the message as a floating-point number representing seconds since the epoch.

set_date(date)
: Set the delivery date of the message to date, a floating-point number representing seconds since the epoch.

get_info()
: Return a string containing the "info" for a message. This is useful for accessing and modifying "info" that is experimental (i.e., not a list of flags).

set_info(info)
: Set "info" to info, which should be a string.

When a MaildirMessage instance is created based upon an mboxMessage or MMDFMessage instance, the Status and X-Status headers are omitted and the following conversions take place:

Resulting state	mboxMessage or MMDFMessage state
"cur" subdirectory	O flag
F flag	F flag
R flag	A flag
S flag	R flag
T flag	D flag

When a MaildirMessage instance is created based upon an MHMessage instance, the following conversions take place:

Resulting state	MHMessage state
"cur" subdirectory	"unseen" sequence
"cur" subdirectory and S flag	no "unseen" sequence
F flag	"flagged" sequence
R flag	"replied" sequence

When a MaildirMessage instance is created based upon a BabylMessage instance, the following conversions take place:

Resulting state	BabylMessage state
"cur" subdirectory	"unseen" label
"cur" subdirectory and S flag	no "unseen" label
P flag	"forwarded" or "resent" label
R flag	"answered" label
T flag	"deleted" label

mboxMessage

class mailbox.mboxMessage([message])
A message with mbox-specific behaviors. Parameter message has the same meaning as with the Message constructor.

Messages in an mbox mailbox are stored together in a single file. The sender's envelope address and the time of delivery are typically stored in a line beginning with "From " that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in Status and X-Status headers.

Conventional flags for mbox messages are as follows:

Flag	Meaning	Explanation
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The "R" and "O" flags are stored in the Status header, and the "D", "F", and "A" flags are stored in the X-Status header. The flags and headers typically appear in the order mentioned.

mboxMessage instances offer the following methods:

get_from()
Return a string representing the "From " line that marks the start of the message in an mbox mailbox. The leading "From " and the trailing newline are excluded.

set_from(from_, time_=None)
Set the "From " line to from_, which should be specified without a leading "From " or trailing newline. For convenience, time_ may be specified and will be formatted appropriately and appended to from_. If time_ is specified, it should be a time.struct_time instance, a tuple suitable for passing to time.strftime(), or True (to use time.gmtime()).

get_flags()
Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of 'R', 'O', 'D', 'F', and 'A'.

set_flags(flags)
Set the flags specified by flags and unset all others. Parameter flags should be the concatenation in any order of zero or more occurrences of each of 'R', 'O', 'D', 'F', and 'A'.

add_flag(flag)
Set the flag(s) specified by flag without changing other flags. To add more than one flag at a time, flag may be a string of more than one character.

remove_flag(flag)
Unset the flag(s) specified by flag without changing other flags. To remove more than one flag at a time, flag maybe a string of more than one character.

When an mboxMessage instance is created based upon a MaildirMessage instance, a "From " line is generated based upon the MaildirMessage instance's delivery date, and the following conversions take place:

Resulting state	MaildirMessage state
R flag	S flag
O flag	"cur" subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an mboxMessage instance is created based upon an MHMessage instance, the following conversions take place:

Resulting state	MHMessage state
R flag and O flag	no "unseen" sequence
O flag	"unseen" sequence
F flag	"flagged" sequence
A flag	"replied" sequence

When an mboxMessage instance is created based upon a BabylMessage instance, the following conversions take place:

Resulting state	BabylMessage state
R flag and O flag	no "unseen" label
O flag	"unseen" label
D flag	"deleted" label
A flag	"answered" label

When a Message instance is created based upon an MMDFMessage instance, the "From " line is copied and all flags directly correspond:

Resulting state	MMDFMessage state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

MHMessage

class mailbox.MHMessage([message])

A message with MH-specific behaviors. Parameter message has the same meaning as with the Message constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard mh and nmh) use sequences in much the same way flags are used with other formats, as follows:

Sequence	Explanation
unseen	Not read, but previously detected by MUA
replied	Replied to
flagged	Marked as important

MHMessage instances offer the following methods:

get_sequences()
Return a list of the names of sequences that include this message.

set_sequences(sequences)
Set the list of sequences that include this message.

add_sequence(sequence)
Add sequence to the list of sequences that include this message.

remove_sequence(sequence)
Remove sequence from the list of sequences that include this message.

When an MHMessage instance is created based upon a MaildirMessage instance, the following conversions take place:

Resulting state	MaildirMessage state
"unseen" sequence	no S flag
"replied" sequence	R flag
"flagged" sequence	F flag

When an MHMessage instance is created based upon an mboxMessage or MMDFMessage instance, the Status and X-Status headers are omitted and the following conversions take place:

Resulting state	mboxMessage or MMDFMessage state
"unseen" sequence	no R flag
"replied" sequence	A flag
"flagged" sequence	F flag

When an MHMessage instance is created based upon a BabylMessage instance, the following conversions take place:

Resulting state	BabylMessage state
"unseen" sequence	"unseen" label
"replied" sequence	"answered" label

BabylMessage

class mailbox.BabylMessage([message])
A message with Babyl-specific behaviors. Parameter message has the same meaning as with the Message constructor.

Certain message labels, called attributes, are defined by convention to have special meanings. The attributes are as follows:

Label	Explanation
unseen	Not read, but previously detected by MUA
deleted	Marked for subsequent deletion
filed	Copied to another file or mailbox
answered	Replied to
forwarded	Forwarded
edited	Modified by the user
resent	Resent

By default, Rmail displays only visible headers. The BabylMessage class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

BabylMessage instances offer the following methods:

get_labels()
: Return a list of labels on the message.

set_labels(labels)
: Set the list of labels on the message to labels.

add_label(label)
: Add label to the list of labels on the message.

remove_label(label)
: Remove label from the list of labels on the message.

get_visible()
: Return an Message instance whose headers are the message's visible headers and whose body is empty.

set_visible(visible)
: Set the message's visible headers to be the same as the headers in message. Parameter visible should be a Message instance, an email.message.Message instance, a string, or a file-like object (which should be open in text mode).

update_visible()
: When a BabylMessage instance's original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows: each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of Date, From, Reply-To, To, CC, and Subject that are present in the original headers but not the visible headers are added to the visible headers.

When a BabylMessage instance is created based upon a MaildirMessage instance, the following conversions take place:

Resulting state	MaildirMessage state
"unseen" label	no S flag
"deleted" label	T flag
"answered" label	R flag
"forwarded" label	P flag

When a BabylMessage instance is created based upon an mboxMessage or MMDFMessage instance, the Status and X-Status headers are omitted and the following conversions take place:

Resulting state	mboxMessage or MMDFMessage state
"unseen" label	no R flag
"deleted" label	D flag
"answered" label	A flag

When a BabylMessage instance is created based upon an MHMessage instance, the following conversions take place:

Resulting state	MHMessage state
"unseen" label	"unseen" sequence
"answered" label	"replied" sequence

MMDFMessage

class mailbox.MMDFMessage([message])

A message with MMDF-specific behaviors. Parameter message has the same meaning as with the Message constructor.

As with message in an mbox mailbox, MMDF messages are stored with the sender's address and the delivery date in an initial line beginning with "From ". Likewise, flags that indicate the state of the message are typically stored in Status and X-Status headers.

Conventional flags for MMDF messages are identical to those of mbox message and are as follows:

Flag	Meaning	Explanation
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The "R" and "O" flags are stored in the Status header, and the "D", "F", and "A" flags are stored in the X-Status header. The flags and headers typically appear in the order mentioned.

MMDFMessage instances offer the following methods, which are identical to those offered by mboxMessage:

get_from()

Return a string representing the "From " line that marks the start of the message in an mbox mailbox. The leading "From " and the trailing newline are excluded.

set_from(from_, time_=None)

Set the "From " line to from_, which should be specified without a leading "From " or trailing newline. For convenience, time_ may be specified and will be formatted appropriately and appended to from_. If time_ is specified, it should be a time.struct_time instance, a tuple suitable for passing to time.strftime(), or True (to use time.gmtime()).

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of 'R', 'O', 'D', 'F', and 'A'.

set_flags(flags)

Set the flags specified by flags and unset all others. Parameter flags should be the concatenation in any order of zero or more occurrences of each of 'R', 'O', 'D', 'F', and 'A'.

add_flag(flag)

Set the flag(s) specified by flag without changing other flags. To add more than one flag at a time, flag may be a string of more than one character.

remove_flag(flag)

Unset the flag(s) specified by flag without changing other flags. To remove more than one flag at a time, flag maybe a string of more than one character.

When an MMDFMessage instance is created based upon a MaildirMessage instance, a "From " line is generated based upon the MaildirMessage instance's delivery date, and the following conversions take place:

Resulting state	MaildirMessage state
R flag	S flag
O flag	"cur" subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an MMDFMessage instance is created based upon an MHMessage instance, the following conversions take place:

Resulting state	MHMessage state
R flag and O flag	no "unseen" sequence
O flag	"unseen" sequence
F flag	"flagged" sequence
A flag	"replied" sequence

When an MMDFMessage instance is created based upon a BabylMessage instance, the following conversions take place:

Resulting state	BabylMessage state
R flag and O flag	no "unseen" label
O flag	"unseen" label
D flag	"deleted" label
A flag	"answered" label

When an MMDFMessage instance is created based upon an mboxMessage instance, the "From " line is copied and all flags directly correspond:

Resulting state	mboxMessage state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

18.4.3 Exceptions

The following exception classes are defined in the mailbox module:

exception mailbox.Error
: The based class for all other module-specific exceptions.

exception mailbox.NoSuchMailboxError
: Raised when a mailbox is expected but is not found, such as when instantiating a Mailbox subclass with a path that does not exist (and with the create parameter set to False), or when opening a folder that does not exist.

exception mailbox.NotEmptyError
: Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

exception mailbox.ExternalClashError
Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely-generated file name already exists.

exception mailbox.FormatError
Raised when the data in a file cannot be parsed, such as when an MH instance attempts to read a corrupted .mh_sequences file.

18.4.4 Deprecated classes and methods

Deprecated since version 2.6.

Older versions of the mailbox module do not support modification of mailboxes, such as adding or removing message, and do not provide classes to represent format-specific message properties. For backward compatibility, the older mailbox classes are still available, but the newer classes should be used in preference to them. The old classes have been removed in Python 3.

Older mailbox objects support only iteration and provide a single public method:

oldmailbox.next()
Return the next message in the mailbox, created with the optional factory argument passed into the mailbox object's constructor. By default this is an rfc822.Message object (see the rfc822 module). Depending on the mailbox implementation the fp attribute of this object may be a true file object or a class instance simulating a file object, taking care of things like message boundaries if multiple mail messages are contained in a single file, etc. If no more messages are available, this method returns None.

Most of the older mailbox classes have names that differ from the current mailbox class names, except for Maildir. For this reason, the new Maildir class defines a next() method and its constructor differs slightly from those of the other new mailbox classes.

The older mailbox classes whose names are not the same as their newer counterparts are as follows:

class mailbox.UnixMailbox(fp[, factory])
Access to a classic Unix-style mailbox, where all messages are contained in a single file and separated by From (a.k.a. From_) lines. The file object fp points to the mailbox file. The optional factory parameter is a callable that should create new message objects. factory is called with one argument, fp by the next() method of the mailbox object. The default is the rfc822.Message class (see the rfc822 module – and the note below).

Note: For reasons of this module's internal implementation, you will probably want to open the fp object in binary mode. This is especially important on Windows.

For maximum portability, messages in a Unix-style mailbox are separated by any line that begins exactly with the string 'From ' (note the trailing space) if preceded by exactly two newlines. Because of the wide-range of variations in practice, nothing else on the From_ line should be considered. However, the current implementation doesn't check for the leading two newlines. This is usually fine for most applications.

The UnixMailbox class implements a more strict version of From_ line checking, using a regular expression that usually correctly matched From_ delimiters. It considers delimiter line to be separated by From name time lines. For maximum portability, use the PortableUnixMailbox class instead. This class is identical to UnixMailbox except that individual messages are separated by only From lines.

class mailbox.PortableUnixMailbox(fp[, factory])
: A less-strict version of UnixMailbox, which considers only the From at the beginning of the line separating messages. The "name time" portion of the From line is ignored, to protect against some variations that are observed in practice. This works since lines in the message which begin with 'From ' are quoted by mail handling software at delivery-time.

class mailbox.MmdfMailbox(fp[, factory])
: Access an MMDF-style mailbox, where all messages are contained in a single file and separated by lines consisting of 4 control-A characters. The file object fp points to the mailbox file. Optional factory is as with the UnixMailbox class.

class mailbox.MHMailbox(dirname[, factory])
: Access an MH mailbox, a directory with each message in a separate file with a numeric name. The name of the mailbox directory is passed in dirname. factory is as with the UnixMailbox class.

class mailbox.BabylMailbox(fp[, factory])
: Access a Babyl mailbox, which is similar to an MMDF mailbox. In Babyl format, each message has two sets of headers, the original headers and the visible headers. The original headers appear before a line containing only '*** EOOH ***' (End-Of-Original-Headers) and the visible headers appear after the EOOH line. Babyl-compliant mail readers will show you only the visible headers, and BabylMailbox objects will return messages containing only the visible headers. You'll have to do your own parsing of the mailbox file to get at the original headers. Mail messages start with the EOOH line and end with a line containing only '\037\014'. factory is as with the UnixMailbox class.

If you wish to use the older mailbox classes with the email module rather than the deprecated rfc822 module, you can do so as follows:

```
import email
import email.Errors
import mailbox

def msgfactory(fp):
    try:
        return email.message_from_file(fp)
    except email.Errors.MessageParseError:
        # Don't return None since that will
        # stop the mailbox iterator
        return ''

mbox = mailbox.UnixMailbox(fp, msgfactory)
```

Alternatively, if you know your mailbox contains only well-formed MIME messages, you can simplify this to:

```
import email
import mailbox

mbox = mailbox.UnixMailbox(fp, email.message_from_file)
```

18.4.5 Examples

A simple example of printing the subjects of all messages in a mailbox that seem interesting:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be None.
```

(continues on next page)

(continued from previous page)

```
    if subject and 'python' in subject.lower():
        print subject
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

This example sorts mail from several mailing lists into different mailboxes, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = dict((name, mailbox.mbox('~/email/%s' % name)) for name in list_names)
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # The message is malformed. Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

18.5 mhlib — Access to MH mailboxes

Deprecated since version 2.6: The mhlib module has been removed in Python 3. Use the mailbox instead.

The mhlib module provides a Python interface to MH folders and their contents.

The module contains three basic classes, MH, which represents a particular collection of folders, Folder, which represents a single folder, and Message, which represents a single message.

class mhlib.MH([path[, profile]])
MH represents a collection of MH folders.

class mhlib.Folder(mh, name)
The Folder class represents a single folder and its messages.

class mhlib.Message(folder, number[, name])
Message objects represent individual messages in a folder. The Message class is derived from mimetools.Message.

18.5.1 MH Objects

MH instances have the following methods:

MH.error(format[, ...])
Print an error message – can be overridden.

MH.getprofile(key)
Return a profile entry (None if not set).

MH.getpath()
Return the mailbox pathname.

MH.getcontext()
Return the current folder name.

MH.setcontext(name)
Set the current folder name.

MH.listfolders()
Return a list of top-level folders.

MH.listallfolders()
Return a list of all folders.

MH.listsubfolders(name)
Return a list of direct subfolders of the given folder.

MH.listallsubfolders(name)
Return a list of all subfolders of the given folder.

MH.makefolder(name)
Create a new folder.

MH.deletefolder(name)
Delete a folder – must have no subfolders.

MH.openfolder(name)
Return a new open folder object.

18.5.2 Folder Objects

Folder instances represent open folders and have the following methods:

Folder.error(format[, ...])
: Print an error message – can be overridden.

Folder.getfullname()
: Return the folder's full pathname.

Folder.getsequencesfilename()
: Return the full pathname of the folder's sequences file.

Folder.getmessagefilename(n)
: Return the full pathname of message n of the folder.

Folder.listmessages()
: Return a list of messages in the folder (as numbers).

Folder.getcurrent()
: Return the current message number.

Folder.setcurrent(n)
: Set the current message number to n.

Folder.parsesequence(seq)
: Parse msgs syntax into list of messages.

Folder.getlast()
: Get last message, or 0 if no messages are in the folder.

Folder.setlast(n)
: Set last message (internal use only).

Folder.getsequences()
: Return dictionary of sequences in folder. The sequence names are used as keys, and the values are the lists of message numbers in the sequences.

Folder.putsequences(dict)
: Return dictionary of sequences in folder name: list.

Folder.removemessages(list)
: Remove messages in list from folder.

Folder.refilemessages(list, tofolder)
: Move messages in list to other folder.

Folder.movemessage(n, tofolder, ton)
: Move one message to a given destination in another folder.

Folder.copymessage(n, tofolder, ton)
: Copy one message to a given destination in another folder.

18.5.3 Message Objects

The Message class adds one method to those of mimetools.Message:

Message.openmessage(n)
: Return a new open message object (costs a file descriptor).

18.6 mimetools — Tools for parsing MIME messages

Deprecated since version 2.3: The email package should be used in preference to the mimetools module. This module is present only to maintain backward compatibility, and it has been removed in 3.x.

This module defines a subclass of the rfc822 module's Message class and a number of utility functions that are useful for the manipulation for MIME multipart or encoded message.

It defines the following items:

class mimetools.Message(fp[, seekable])
: Return a new instance of the Message class. This is a subclass of the rfc822.Message class, with some additional methods (see below). The seekable argument has the same meaning as for rfc822.Message.

mimetools.choose_boundary()
: Return a unique string that has a high likelihood of being usable as a part boundary. The string has the form `'hostipaddr.uid.pid.timestamp.random'`.

mimetools.decode(input, output, encoding)
: Read data encoded using the allowed MIME encoding from open file object input and write the decoded data to open file object output. Valid values for encoding include `'base64'`, `'quoted-printable'`, `'uuencode'`, `'x-uuencode'`, `'uue'`, `'x-uue'`, `'7bit'`, and `'8bit'`. Decoding messages encoded in `'7bit'` or `'8bit'` has no effect. The input is simply copied to the output.

mimetools.encode(input, output, encoding)
: Read data from open file object input and write it encoded using the allowed MIME encoding to open file object output. Valid values for encoding are the same as for decode().

mimetools.copyliteral(input, output)
: Read lines from open file input until EOF and write them to open file output.

mimetools.copybinary(input, output)
: Read blocks until EOF from open file input and write them to open file output. The block size is currently fixed at 8192.

See also:

Module email Comprehensive email handling package; supersedes the mimetools module.

Module rfc822 Provides the base class for mimetools.Message.

Module multifile Support for reading files which contain distinct parts, such as MIME data.

http://faqs.cs.uu.nl/na-dir/mail/mime-faq/.html The MIME Frequently Asked Questions document. For an overview of MIME, see the answer to question 1.1 in Part 1 of this document.

18.6.1 Additional Methods of Message Objects

The Message class defines the following methods in addition to the rfc822.Message methods:

Message.getplist()
: Return the parameter list of the Content-Type header. This is a list of strings. For parameters of the form key=value, key is converted to lower case but value is not. For example, if the message contains the header Content-type: text/html; spam=1; Spam=2; Spam then getplist() will return the Python list `['spam=1', 'spam=2', 'Spam']`.

Message.getparam(name)
: Return the value of the first parameter (as returned by getplist()) of the form name=value for the given name. If value is surrounded by quotes of the form '<...>' or '"..."', these are removed.

Message.getencoding()
: Return the encoding specified in the Content-Transfer-Encoding message header. If no such header exists, return `'7bit'`. The encoding is converted to lower case.

Message.gettype()
: Return the message type (of the form type/subtype) as specified in the Content-Type header. If no such header exists, return `'text/plain'`. The type is converted to lower case.

Message.getmaintype()
: Return the main type as specified in the Content-Type header. If no such header exists, return `'text'`. The main type is converted to lower case.

Message.getsubtype()
: Return the subtype as specified in the Content-Type header. If no such header exists, return `'plain'`. The subtype is converted to lower case.

18.7 mimetypes — Map filenames to MIME types

Source code: Lib/mimetypes.py

The mimetypes module converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension; encodings are not supported for the latter conversion.

The module provides one class and a number of convenience functions. The functions are the normal interface to this module, but some applications may be interested in the class as well.

The functions described below provide the primary interface for this module. If the module has not been initialized, they will call init() if they rely on the information init() sets up.

mimetypes.guess_type(url, strict=True)
: Guess the type of a file based on its filename or URL, given by url. The return value is a tuple (type, encoding) where type is None if the type can't be guessed (missing or unknown suffix) or a string of the form `'type/subtype'`, usable for a MIME content-type header.

  encoding is None for no encoding or the name of the program used to encode (e.g. compress or gzip). The encoding is suitable for use as a Content-Encoding header, not as a Content-Transfer-Encoding header. The mappings are table driven. Encoding suffixes are case sensitive; type suffixes are first tried case sensitively, then case insensitively.

  The optional strict argument is a flag specifying whether the list of known MIME types is limited to only the official types registered with IANA. When strict is True (the default), only the IANA types are supported; when strict is False, some additional non-standard but commonly used MIME types are also recognized.

mimetypes.guess_all_extensions(type, strict=True)
: Guess the extensions for a file based on its MIME type, given by type. The return value is a list of strings giving all possible filename extensions, including the leading dot (`'.'`). The extensions are not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type type by guess_type().

  The optional strict argument has the same meaning as with the guess_type() function.

mimetypes.guess_extension(type, strict=True)
: Guess the extension for a file based on its MIME type, given by type. The return value is a string giving a filename extension, including the leading dot (`'.'`). The extension is not guaranteed to have

been associated with any particular data stream, but would be mapped to the MIME type type by guess_type(). If no extension can be guessed for type, None is returned.

The optional strict argument has the same meaning as with the guess_type() function.

Some additional functions and data items are available for controlling the behavior of the module.

mimetypes.init(files=None)
Initialize the internal data structures. If given, files must be a sequence of file names which should be used to augment the default type map. If omitted, the file names to use are taken from knownfiles; on Windows, the current registry settings are loaded. Each file named in files or knownfiles takes precedence over those named before it. Calling init() repeatedly is allowed.

Specifying an empty list for files will prevent the system defaults from being applied: only the well-known values will be present from a built-in list.

Changed in version 2.7: Previously, Windows registry settings were ignored.

mimetypes.read_mime_types(filename)
Load the type map given in the file filename, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot ('.'), to strings of the form 'type/subtype'. If the file filename does not exist or cannot be read, None is returned.

mimetypes.add_type(type, ext, strict=True)
Add a mapping from the MIME type type to the extension ext. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When strict is True (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

mimetypes.inited
Flag indicating whether or not the global data structures have been initialized. This is set to True by init().

mimetypes.knownfiles
List of type map file names commonly installed. These files are typically named mime.types and are installed in different locations by different packages.

mimetypes.suffix_map
Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the .tgz extension is mapped to .tar.gz to allow the encoding and type to be recognized separately.

mimetypes.encodings_map
Dictionary mapping filename extensions to encoding types.

mimetypes.types_map
Dictionary mapping filename extensions to MIME types.

mimetypes.common_types
Dictionary mapping filename extensions to non-standard, but commonly found MIME types.

An example usage of the module:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
```

(continues on next page)

(continued from previous page)

```
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

18.7.1 MimeTypes Objects

The MimeTypes class may be useful for applications which may want more than one MIME-type database; it provides an interface similar to the one of the mimetypes module.

class mimetypes.MimeTypes(filenames=(), strict=True)
: This class represents a MIME-types database. By default, it provides access to the same database as the rest of this module. The initial database is a copy of that provided by the module, and may be extended by loading additional mime.types-style files into the database using the read() or readfp() methods. The mapping dictionaries may also be cleared before loading additional data if the default data is not desired.

  The optional filenames parameter can be used to cause additional files to be loaded "on top" of the default database.

  suffix_map
  : Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the .tgz extension is mapped to .tar.gz to allow the encoding and type to be recognized separately. This is initially a copy of the global suffix_map defined in the module.

  encodings_map
  : Dictionary mapping filename extensions to encoding types. This is initially a copy of the global encodings_map defined in the module.

  types_map
  : Tuple containing two dictionaries, mapping filename extensions to MIME types: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by common_types and types_map.

  types_map_inv
  : Tuple containing two dictionaries, mapping MIME types to a list of filename extensions: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by common_types and types_map.

  guess_extension(type, strict=True)
  : Similar to the guess_extension() function, using the tables stored as part of the object.

  guess_type(url, strict=True)
  : Similar to the guess_type() function, using the tables stored as part of the object.

  guess_all_extensions(type, strict=True)
  : Similar to the guess_all_extensions() function, using the tables stored as part of the object.

  read(filename, strict=True)
  : Load MIME information from a file named filename. This uses readfp() to parse the file.

    If strict is True, information will be added to list of standard types, else to the list of non-standard types.

  readfp(fp, strict=True)
  : Load MIME type information from an open file fp. The file must have the format of the standard mime.types files.

If strict is True, information will be added to the list of standard types, else to the list of non-standard types.

read_windows_registry(strict=True)
Load MIME type information from the Windows registry. Availability: Windows.

If strict is True, information will be added to the list of standard types, else to the list of non-standard types.

New in version 2.7.

18.8 MimeWriter — Generic MIME file writer

Deprecated since version 2.3: The email package should be used in preference to the MimeWriter module. This module is present only to maintain backward compatibility.

This module defines the class MimeWriter. The MimeWriter class implements a basic formatter for creating MIME multi-part files. It doesn't seek around the output file nor does it use large amounts of buffer space. You must write the parts out in the order that they should occur in the final file. MimeWriter does buffer the headers you add, allowing you to rearrange their order.

class MimeWriter.MimeWriter(fp)
Return a new instance of the MimeWriter class. The only argument passed, fp, is a file object to be used for writing. Note that a StringIO object could also be used.

18.8.1 MimeWriter Objects

MimeWriter instances have the following methods:

MimeWriter.addheader(key, value[, prefix])
Add a header line to the MIME message. The key is the name of the header, where the value obviously provides the value of the header. The optional argument prefix determines where the header is inserted; 0 means append at the end, 1 is insert at the start. The default is to append.

MimeWriter.flushheaders()
Causes all headers accumulated so far to be written out (and forgotten). This is useful if you don't need a body part at all, e.g. for a subpart of type message/rfc822 that's (mis)used to store some header-like information.

MimeWriter.startbody(ctype[, plist[, prefix]])
Returns a file-like object which can be used to write to the body of the message. The content-type is set to the provided ctype, and the optional parameter plist provides additional parameters for the content-type declaration. prefix functions as in addheader() except that the default is to insert at the start.

MimeWriter.startmultipartbody(subtype[, boundary[, plist[, prefix]]])
Returns a file-like object which can be used to write to the body of the message. Additionally, this method initializes the multi-part code, where subtype provides the multipart subtype, boundary may provide a user-defined boundary specification, and plist provides optional parameters for the subtype. prefix functions as in startbody(). Subparts should be created using nextpart().

MimeWriter.nextpart()
Returns a new instance of MimeWriter which represents an individual part in a multipart message. This may be used to write the part as well as used for creating recursively complex multipart messages. The message must first be initialized with startmultipartbody() before using nextpart().

MimeWriter.lastpart()
: This is used to designate the last part of a multipart message, and should always be used when writing multipart messages.

18.9 mimify — MIME processing of mail messages

Deprecated since version 2.3: The email package should be used in preference to the mimify module. This module is present only to maintain backward compatibility.

The mimify module defines two functions to convert mail messages to and from MIME format. The mail message can be either a simple message or a so-called multipart message. Each part is treated separately. Mimifying (a part of) a message entails encoding the message as quoted-printable if it contains any characters that cannot be represented using 7-bit ASCII. Unmimifying (a part of) a message entails undoing the quoted-printable encoding. Mimify and unmimify are especially useful when a message has to be edited before being sent. Typical use would be:

```
unmimify message
edit message
mimify message
send message
```

The modules defines the following user-callable functions and user-settable variables:

mimify.mimify(infile, outfile)
: Copy the message in infile to outfile, converting parts to quoted-printable and adding MIME mail headers when necessary. infile and outfile can be file objects (actually, any object that has a readline() method (for infile) or a write() method (for outfile)) or strings naming the files. If infile and outfile are both strings, they may have the same value.

mimify.unmimify(infile, outfile[, decode_base64])
: Copy the message in infile to outfile, decoding all quoted-printable parts. infile and outfile can be file objects (actually, any object that has a readline() method (for infile) or a write() method (for outfile)) or strings naming the files. If infile and outfile are both strings, they may have the same value. If the decode_base64 argument is provided and tests true, any parts that are coded in the base64 encoding are decoded as well.

mimify.mime_decode_header(line)
: Return a decoded version of the encoded header line in line. This only supports the ISO 8859-1 charset (Latin-1).

mimify.mime_encode_header(line)
: Return a MIME-encoded version of the header line in line.

mimify.MAXLEN
: By default, a part will be encoded as quoted-printable when it contains any non-ASCII characters (characters with the 8th bit set), or if there are any lines longer than MAXLEN characters (default value 200).

mimify.CHARSET
: When not specified in the mail headers, a character set must be filled in. The string used is stored in CHARSET, and the default value is ISO-8859-1 (also known as Latin1 (latin-one)).

This module can also be used from the command line. Usage is as follows:

```
mimify.py -e [-l length] [infile [outfile]]
mimify.py -d [-b] [infile [outfile]]
```

to encode (mimify) and decode (unmimify) respectively. infile defaults to standard input, outfile defaults to standard output. The same file can be specified for input and output.

If the -l option is given when encoding, if there are any lines longer than the specified length, the containing part will be encoded.

If the -b option is given when decoding, any base64 parts will be decoded as well.

See also:

Module quopri Encode and decode MIME quoted-printable files.

18.10 multifile — Support for files containing distinct parts

Deprecated since version 2.5: The email package should be used in preference to the multifile module. This module is present only to maintain backward compatibility.

The MultiFile object enables you to treat sections of a text file as file-like input objects, with '' being returned by readline() when a given delimiter pattern is encountered. The defaults of this class are designed to make it useful for parsing MIME multipart messages, but by subclassing it and overriding methods it can be easily adapted for more general use.

class multifile.MultiFile(fp[, seekable])
Create a multi-file. You must instantiate this class with an input object argument for the MultiFile instance to get lines from, such as a file object returned by open().

MultiFile only ever looks at the input object's readline(), seek() and tell() methods, and the latter two are only needed if you want random access to the individual MIME parts. To use MultiFile on a non-seekable stream object, set the optional seekable argument to false; this will prevent using the input object's seek() and tell() methods.

It will be useful to know that in MultiFile's view of the world, text is composed of three kinds of lines: data, section-dividers, and end-markers. MultiFile is designed to support parsing of messages that may have multiple nested message parts, each with its own pattern for section-divider and end-marker lines.

See also:

Module email Comprehensive email handling package; supersedes the multifile module.

18.10.1 MultiFile Objects

A MultiFile instance has the following methods:

MultiFile.readline(str)
Read a line. If the line is data (not a section-divider or end-marker or real EOF) return it. If the line matches the most-recently-stacked boundary, return '' and set self.last to 1 or 0 according as the match is or is not an end-marker. If the line matches any other stacked boundary, raise an error. On encountering end-of-file on the underlying stream object, the method raises Error unless all boundaries have been popped.

MultiFile.readlines(str)
Return all lines remaining in this part as a list of strings.

MultiFile.read()
Read all lines, up to the next section. Return them as a single (multiline) string. Note that this doesn't take a size argument!

MultiFile.seek(pos[, whence])
: Seek. Seek indices are relative to the start of the current section. The pos and whence arguments are interpreted as for a file seek.

MultiFile.tell()
: Return the file position relative to the start of the current section.

MultiFile.next()
: Skip lines to the next section (that is, read lines until a section-divider or end-marker has been consumed). Return true if there is such a section, false if an end-marker is seen. Re-enable the most-recently-pushed boundary.

MultiFile.is_data(str)
: Return true if str is data and false if it might be a section boundary. As written, it tests for a prefix other than `'--'` at start of line (which all MIME boundaries have) but it is declared so it can be overridden in derived classes.

  Note that this test is used intended as a fast guard for the real boundary tests; if it always returns false it will merely slow processing, not cause it to fail.

MultiFile.push(str)
: Push a boundary string. When a decorated version of this boundary is found as an input line, it will be interpreted as a section-divider or end-marker (depending on the decoration, see RFC 2045). All subsequent reads will return the empty string to indicate end-of-file, until a call to pop() removes the boundary a or next() call reenables it.

  It is possible to push more than one boundary. Encountering the most-recently-pushed boundary will return EOF; encountering any other boundary will raise an error.

MultiFile.pop()
: Pop a section boundary. This boundary will no longer be interpreted as EOF.

MultiFile.section_divider(str)
: Turn a boundary into a section-divider line. By default, this method prepends `'--'` (which MIME section boundaries have) but it is declared so it can be overridden in derived classes. This method need not append LF or CR-LF, as comparison with the result ignores trailing whitespace.

MultiFile.end_marker(str)
: Turn a boundary string into an end-marker line. By default, this method prepends `'--'` and appends `'--'` (like a MIME-multipart end-of-message marker) but it is declared so it can be overridden in derived classes. This method need not append LF or CR-LF, as comparison with the result ignores trailing whitespace.

Finally, MultiFile instances have two public instance variables:

MultiFile.level
: Nesting depth of the current part.

MultiFile.last
: True if the last end-of-file was for an end-of-message marker.

18.10.2 MultiFile Example

```
import mimetools
import multifile
import StringIO

def extract_mime_part_matching(stream, mimetype):
```

(continues on next page)

(continued from previous page)

```
    """Return the first element in a multipart MIME message on stream
    matching mimetype."""

    msg = mimetools.Message(stream)
    msgtype = msg.gettype()
    params = msg.getplist()

    data = StringIO.StringIO()
    if msgtype[:10] == "multipart/":

        file = multifile.MultiFile(stream)
        file.push(msg.getparam("boundary"))
        while file.next():
            submsg = mimetools.Message(file)
            try:
                data = StringIO.StringIO()
                mimetools.decode(file, data, submsg.getencoding())
            except ValueError:
                continue
            if submsg.gettype() == mimetype:
                break
        file.pop()
    return data.getvalue()
```

18.11 rfc822 — Parse RFC 2822 mail headers

Deprecated since version 2.3: The email package should be used in preference to the rfc822 module. This module is present only to maintain backward compatibility, and has been removed in Python 3.

This module defines a class, Message, which represents an "email message" as defined by the Internet standard RFC 2822.[1] Such messages consist of a collection of message headers, and a message body. This module also defines a helper class AddressList for parsing RFC 2822 addresses. Please refer to the RFC for information on the specific syntax of RFC 2822 messages.

The mailbox module provides classes to read mailboxes produced by various end-user mail programs.

class rfc822.Message(file[, seekable])

A Message instance is instantiated with an input object as parameter. Message relies only on the input object having a readline() method; in particular, ordinary file objects qualify. Instantiation reads headers from the input object up to a delimiter line (normally a blank line) and stores them in the instance. The message body, following the headers, is not consumed.

This class can work with any input object that supports a readline() method. If the input object has seek and tell capability, the rewindbody() method will work; also, illegal lines will be pushed back onto the input stream. If the input object lacks seek but has an unread() method that can push back a line of input, Message will use that to push back illegal lines. Thus this class can be used to parse messages coming from a buffered stream.

The optional seekable argument is provided as a workaround for certain stdio libraries in which tell() discards buffered data before discovering that the lseek() system call doesn't work. For maximum portability, you should set the seekable argument to zero to prevent that initial tell() when passing in an unseekable object such as a file object created from a socket object.

[1] This module originally conformed to RFC 822, hence the name. Since then, RFC 2822 has been released as an update to RFC 822. This module should be considered RFC 2822-conformant, especially in cases where the syntax or semantics have changed since RFC 822.

Input lines as read from the file may either be terminated by CR-LF or by a single linefeed; a terminating CR-LF is replaced by a single linefeed before the line is stored.

All header matching is done independent of upper or lower case; e.g. m['From'], m['from'] and m['FROM'] all yield the same result.

class rfc822.AddressList(field)
You may instantiate the AddressList helper class using a single string parameter, a comma-separated list of RFC 2822 addresses to be parsed. (The parameter None yields an empty list.)

rfc822.quote(str)
Return a new string with backslashes in str replaced by two backslashes and double quotes replaced by backslash-double quote.

rfc822.unquote(str)
Return a new string which is an unquoted version of str. If str ends and begins with double quotes, they are stripped off. Likewise if str ends and begins with angle brackets, they are stripped off.

rfc822.parseaddr(address)
Parse address, which should be the value of some address-containing field such as To or Cc, into its constituent "realname" and "email address" parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple (None, None) is returned.

rfc822.dump_address_pair(pair)
The inverse of parseaddr(), this takes a 2-tuple of the form (realname, email_address) and returns the string value suitable for a To or Cc header. If the first element of pair is false, then the second element is returned unmodified.

rfc822.parsedate(date)
Attempts to parse a date according to the rules in RFC 2822. however, some mailers don't follow that format as specified, so parsedate() tries to guess correctly in such cases. date is a string containing an RFC 2822 date, such as 'Mon, 20 Nov 1995 19:12:08 -0500'. If it succeeds in parsing the date, parsedate() returns a 9-tuple that can be passed directly to time.mktime(); otherwise None will be returned. Note that indexes 6, 7, and 8 of the result tuple are not usable.

rfc822.parsedate_tz(date)
Performs the same function as parsedate(), but returns either None or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to time.mktime(), and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time). (Note that the sign of the timezone offset is the opposite of the sign of the time.timezone variable for the same timezone; the latter variable follows the POSIX standard while this module follows RFC 2822.) If the input string has no timezone, the last element of the tuple returned is None. Note that indexes 6, 7, and 8 of the result tuple are not usable.

rfc822.mktime_tz(tuple)
Turn a 10-tuple as returned by parsedate_tz() into a UTC timestamp. If the timezone item in the tuple is None, assume local time. Minor deficiency: this first interprets the first 8 elements as a local time and then compensates for the timezone difference; this may yield a slight error around daylight savings time switch dates. Not enough to worry about for common use.

See also:

Module email Comprehensive email handling package; supersedes the rfc822 module.

Module mailbox Classes to read various mailbox formats produced by end-user mail programs.

Module mimetools Subclass of rfc822.Message that handles MIME encoded messages.

18.11.1 Message Objects

A Message instance has the following methods:

Message.rewindbody()
Seek to the start of the message body. This only works if the file object is seekable.

Message.isheader(line)
Returns a line's canonicalized fieldname (the dictionary key that will be used to index it) if the line is a legal RFC 2822 header; otherwise returns None (implying that parsing should stop here and the line be pushed back on the input stream). It is sometimes useful to override this method in a subclass.

Message.islast(line)
Return true if the given line is a delimiter on which Message should stop. The delimiter line is consumed, and the file object's read location positioned immediately after it. By default this method just checks that the line is blank, but you can override it in a subclass.

Message.iscomment(line)
Return True if the given line should be ignored entirely, just skipped. By default this is a stub that always returns False, but you can override it in a subclass.

Message.getallmatchingheaders(name)
Return a list of lines consisting of all headers matching name, if any. Each physical line, whether it is a continuation line or not, is a separate list item. Return the empty list if no header matches name.

Message.getfirstmatchingheader(name)
Return a list of lines comprising the first header matching name, and its continuation line(s), if any. Return None if there is no header matching name.

Message.getrawheader(name)
Return a single string consisting of the text after the colon in the first header matching name. This includes leading whitespace, the trailing linefeed, and internal linefeeds and whitespace if there any continuation line(s) were present. Return None if there is no header matching name.

Message.getheader(name[, default])
Return a single string consisting of the last header matching name, but strip leading and trailing whitespace. Internal whitespace is not stripped. The optional default argument can be used to specify a different default to be returned when there is no header matching name; it defaults to None. This is the preferred way to get parsed headers.

Message.get(name[, default])
An alias for getheader(), to make the interface more compatible with regular dictionaries.

Message.getaddr(name)
Return a pair (full name, email address) parsed from the string returned by getheader(name). If no header matching name exists, return (None, None); otherwise both the full name and the address are (possibly empty) strings.

Example: If m's first From header contains the string 'jack@cwi.nl (Jack Jansen)', then m.getaddr('From') will yield the pair ('Jack Jansen', 'jack@cwi.nl'). If the header contained 'Jack Jansen <jack@cwi.nl>' instead, it would yield the exact same result.

Message.getaddrlist(name)
This is similar to getaddr(list), but parses a header containing a list of email addresses (e.g. a To header) and returns a list of (full name, email address) pairs (even if there was only one address in the header). If there is no header matching name, return an empty list.

If multiple headers exist that match the named header (e.g. if there are several Cc headers), all are parsed for addresses. Any continuation lines the named headers contain are also parsed.

Message.getdate(name)
Retrieve a header using getheader() and parse it into a 9-tuple compatible with time.mktime(); note that fields 6, 7, and 8 are not usable. If there is no header matching name, or it is unparsable, return None.

Date parsing appears to be a black art, and not all mailers adhere to the standard. While it has been tested and found correct on a large collection of email from many sources, it is still possible that this function may occasionally yield an incorrect result.

Message.getdate_tz(name)
Retrieve a header using getheader() and parse it into a 10-tuple; the first 9 elements will make a tuple compatible with time.mktime(), and the 10th is a number giving the offset of the date's timezone from UTC. Note that fields 6, 7, and 8 are not usable. Similarly to getdate(), if there is no header matching name, or it is unparsable, return None.

Message instances also support a limited mapping interface. In particular: m[name] is like m.getheader(name) but raises KeyError if there is no matching header; and len(m), m.get(name[, default]), name in m, m.keys(), m.values() m.items(), and m.setdefault(name[, default]) act as expected, with the one difference that setdefault() uses an empty string as the default value. Message instances also support the mapping writable interface m[name] = value and del m[name]. Message objects do not support the clear(), copy(), popitem(), or update() methods of the mapping interface. (Support for get() and setdefault() was only added in Python 2.2.)

Finally, Message instances have some public instance variables:

Message.headers
A list containing the entire set of header lines, in the order in which they were read (except that setitem calls may disturb this order). Each line contains a trailing newline. The blank line terminating the headers is not contained in the list.

Message.fp
The file or file-like object passed at instantiation time. This can be used to read the message content.

Message.unixfrom
The Unix From line, if the message had one, or an empty string. This is needed to regenerate the message in some contexts, such as an mbox-style mailbox file.

18.11.2 AddressList Objects

An AddressList instance has the following methods:

AddressList.__len__()
Return the number of addresses in the address list.

AddressList.__str__()
Return a canonicalized string representation of the address list. Addresses are rendered in "name" <host@domain> form, comma-separated.

AddressList.__add__(alist)
Return a new AddressList instance that contains all addresses in both AddressList operands, with duplicates removed (set union).

AddressList.__iadd__(alist)
In-place version of __add__(); turns this AddressList instance into the union of itself and the right-hand instance, alist.

AddressList.__sub__(alist)
Return a new AddressList instance that contains every address in the left-hand AddressList operand that is not present in the right-hand address operand (set difference).

AddressList.__isub__(alist)
In-place version of __sub__(), removing addresses in this list which are also in alist.

Finally, AddressList instances have one public instance variable:

AddressList.addresslist
A list of tuple string pairs, one per address. In each member, the first is the canonicalized name part, the second is the actual route-address ('@'-separated username-host.domain pair).

18.12 base64 — RFC 3548: Base16, Base32, Base64 Data Encodings

This module provides data encoding and decoding as specified in RFC 3548. This standard defines the Base16, Base32, and Base64 algorithms for encoding and decoding arbitrary binary strings into text strings that can be safely sent by email, used as parts of URLs, or included as part of an HTTP POST request. The encoding algorithm is not the same as the uuencode program.

There are two interfaces provided by this module. The modern interface supports encoding and decoding string objects using both base-64 alphabets defined in RFC 3548 (normal, and URL- and filesystem-safe). The legacy interface provides for encoding and decoding to and from file-like objects as well as strings, but only using the Base64 standard alphabet.

The modern interface, which was introduced in Python 2.4, provides:

base64.b64encode(s[, altchars])
Encode a string using Base64.

s is the string to encode. Optional altchars must be a string of at least length 2 (additional characters are ignored) which specifies an alternative alphabet for the + and / characters. This allows an application to e.g. generate URL or filesystem safe Base64 strings. The default is None, for which the standard Base64 alphabet is used.

The encoded string is returned.

base64.b64decode(s[, altchars])
Decode a Base64 encoded string.

s is the string to decode. Optional altchars must be a string of at least length 2 (additional characters are ignored) which specifies the alternative alphabet used instead of the + and / characters.

The decoded string is returned. A TypeError is raised if s is incorrectly padded. Characters that are neither in the normal base-64 alphabet nor the alternative alphabet are discarded prior to the padding check.

base64.standard_b64encode(s)
Encode string s using the standard Base64 alphabet.

base64.standard_b64decode(s)
Decode string s using the standard Base64 alphabet.

base64.urlsafe_b64encode(s)
Encode string s using the URL- and filesystem-safe alphabet, which substitutes - instead of + and _ instead of / in the standard Base64 alphabet. The result can still contain =.

base64.urlsafe_b64decode(s)
Decode string s using the URL- and filesystem-safe alphabet, which substitutes - instead of + and _ instead of / in the standard Base64 alphabet.

base64.b32encode(s)
Encode a string using Base32. s is the string to encode. The encoded string is returned.

base64.b32decode(s[, casefold[, map01]])
Decode a Base32 encoded string.

s is the string to decode. Optional casefold is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is False.

RFC 3548 allows for optional mapping of the digit 0 (zero) to the letter O (oh), and for optional mapping of the digit 1 (one) to either the letter I (eye) or letter L (el). The optional argument map01 when not None, specifies which letter the digit 1 should be mapped to (when map01 is not None, the digit 0 is always mapped to the letter O). For security purposes the default is None, so that 0 and 1 are not allowed in the input.

The decoded string is returned. A TypeError is raised if s is incorrectly padded or if there are non-alphabet characters present in the string.

base64.b16encode(s)
Encode a string using Base16.

s is the string to encode. The encoded string is returned.

base64.b16decode(s[, casefold])
Decode a Base16 encoded string.

s is the string to decode. Optional casefold is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is False.

The decoded string is returned. A TypeError is raised if s were incorrectly padded or if there are non-alphabet characters present in the string.

The legacy interface:

base64.decode(input, output)
Decode the contents of the input file and write the resulting binary data to the output file. input and output must either be file objects or objects that mimic the file object interface. input will be read until input.read() returns an empty string.

base64.decodestring(s)
Decode the string s, which must contain one or more lines of base64 encoded data, and return a string containing the resulting binary data.

base64.encode(input, output)
Encode the contents of the input file and write the resulting base64 encoded data to the output file. input and output must either be file objects or objects that mimic the file object interface. input will be read until input.read() returns an empty string. encode() returns the encoded data plus a trailing newline character ('\n').

base64.encodestring(s)
Encode the string s, which can contain arbitrary binary data, and return a string containing one or more lines of base64-encoded data. encodestring() returns a string containing one or more lines of base64-encoded data always including an extra trailing newline ('\n').

An example usage of the module:

```
>>> import base64
>>> encoded = base64.b64encode('data to be encoded')
>>> encoded
'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
'data to be encoded'
```

See also:

Module binascii Support module containing ASCII-to-binary and binary-to-ASCII conversions.

RFC 1521 - MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Form
Section 5.2, "Base64 Content-Transfer-Encoding," provides the definition of the base64 encoding.

18.13 binhex — Encode and decode binhex4 files

This module encodes and decodes files in binhex4 format, a format allowing representation of Macintosh files in ASCII. On the Macintosh, both forks of a file and the finder information are encoded (or decoded), on other platforms only the data fork is handled.

Note: In Python 3.x, special Macintosh support has been removed.

The binhex module defines the following functions:

binhex.binhex(input, output)
Convert a binary file with filename input to binhex file output. The output parameter can either be a filename or a file-like object (any object supporting a write() and close() method).

binhex.hexbin(input[, output])
Decode a binhex file input. input may be a filename or a file-like object supporting read() and close() methods. The resulting file is written to a file named output, unless the argument is omitted in which case the output filename is read from the binhex file.

The following exception is also defined:

exception binhex.Error
Exception raised when something can't be encoded using the binhex format (for example, a filename is too long to fit in the filename field), or when input is not properly encoded binhex data.

See also:

Module binascii Support module containing ASCII-to-binary and binary-to-ASCII conversions.

18.13.1 Notes

There is an alternative, more powerful interface to the coder and decoder, see the source for details.

If you code or decode textfiles on non-Macintosh platforms they will still use the old Macintosh newline convention (carriage-return as end of line).

As of this writing, hexbin() appears to not work in all cases.

18.14 binascii — Convert between binary and ASCII

The binascii module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like uu, base64, or binhex instead. The binascii module contains low-level functions written in C for greater speed that are used by the higher-level modules.

The binascii module defines the following functions:

binascii.a2b_uu(string)
: Convert a single line of uuencoded data back to binary and return the binary data. Lines normally contain 45 (binary) bytes, except for the last line. Line data may be followed by whitespace.

binascii.b2a_uu(data)
: Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of data should be at most 45.

binascii.a2b_base64(string)
: Convert a block of base64 data back to binary and return the binary data. More than one line may be passed at a time.

binascii.b2a_base64(data)
: Convert binary data to a line of ASCII characters in base64 coding. The return value is the converted line, including a newline char. The newline is added because the original use case for this function was to feed it a series of 57 byte input lines to get output lines that conform to the MIME-base64 standard. Otherwise the output conforms to RFC 3548.

binascii.a2b_qp(string[, header])
: Convert a block of quoted-printable data back to binary and return the binary data. More than one line may be passed at a time. If the optional argument header is present and true, underscores will be decoded as spaces.

binascii.b2a_qp(data[, quotetabs, istext, header])
: Convert binary data to a line(s) of ASCII characters in quoted-printable encoding. The return value is the converted line(s). If the optional argument quotetabs is present and true, all tabs and spaces will be encoded. If the optional argument istext is present and true, newlines are not encoded but trailing whitespace will be encoded. If the optional argument header is present and true, spaces will be encoded as underscores per RFC1522. If the optional argument header is present and false, newline characters will be encoded as well; otherwise linefeed conversion might corrupt the binary data stream.

binascii.a2b_hqx(string)
: Convert binhex4 formatted ASCII data to binary, without doing RLE-decompression. The string should contain a complete number of binary bytes, or (in case of the last portion of the binhex4 data) have the remaining bits zero.

binascii.rledecode_hqx(data)
: Perform RLE-decompression on the data, as per the binhex4 standard. The algorithm uses 0x90 after a byte as a repeat indicator, followed by a count. A count of 0 specifies a byte value of 0x90. The routine returns the decompressed data, unless data input data ends in an orphaned repeat indicator, in which case the Incomplete exception is raised.

binascii.rlecode_hqx(data)
: Perform binhex4 style RLE-compression on data and return the result.

binascii.b2a_hqx(data)
: Perform hexbin4 binary-to-ASCII translation and return the resulting string. The argument should already be RLE-coded, and have a length divisible by 3 (except possibly the last fragment).

binascii.crc_hqx(data, crc)
: Compute a 16-bit CRC value of data, starting with an initial crc and returning the result. This uses the CRC-CCITT polynomial $x^{16} + x^{12} + x^5 + 1$, often represented as 0x1021. This CRC is used in the binhex4 format.

binascii.crc32(data[, crc])
: Compute CRC-32, the 32-bit checksum of data, starting with an initial crc. This is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```
print binascii.crc32("hello world")
# Or, in two pieces:
crc = binascii.crc32("hello")
crc = binascii.crc32(" world", crc) & 0xffffffff
print 'crc32 = 0x%08x' % crc
```

Note: To generate the same numeric value across all Python versions and platforms use crc32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

Changed in version 2.6: The return value is in the range [-2**31, 2**31-1] regardless of platform. In the past the value would be signed on some platforms and unsigned on others. Use & 0xffffffff on the value if you want it to match Python 3 behavior.

Changed in version 3.0: The return value is unsigned and in the range [0, 2**32-1] regardless of platform.

binascii.b2a_hex(data)
binascii.hexlify(data)
: Return the hexadecimal representation of the binary data. Every byte of data is converted into the corresponding 2-digit hex representation. The resulting string is therefore twice as long as the length of data.

binascii.a2b_hex(hexstr)
binascii.unhexlify(hexstr)
: Return the binary data represented by the hexadecimal string hexstr. This function is the inverse of b2a_hex(). hexstr must contain an even number of hexadecimal digits (which can be upper or lower case), otherwise a TypeError is raised.

exception binascii.Error
: Exception raised on errors. These are usually programming errors.

exception binascii.Incomplete
: Exception raised on incomplete data. These are usually not programming errors, but may be handled by reading a little more data and trying again.

See also:

Module base64 Support for RFC compliant base64-style encoding in base 16, 32, and 64.

Module binhex Support for the binhex format used on the Macintosh.

Module uu Support for UU encoding used on Unix.

Module quopri Support for quoted-printable encoding used in MIME email messages.

18.15 quopri — Encode and decode MIME quoted-printable data

Source code: Lib/quopri.py

This module performs quoted-printable transport encoding and decoding, as defined in RFC 1521: "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies". The quoted-printable encoding is designed for data where there are relatively few nonprintable characters; the base64 encoding scheme available via the base64 module is more compact if there are many such characters, as when sending a graphics file.

quopri.decode(input, output[, header])
: Decode the contents of the input file and write the resulting decoded binary data to the output file. input and output must either be file objects or objects that mimic the file object interface. input will be read until input.readline() returns an empty string. If the optional argument header is present and true, underscore will be decoded as space. This is used to decode "Q"-encoded headers as described in RFC 1522: "MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text".

quopri.encode(input, output, quotetabs)
: Encode the contents of the input file and write the resulting quoted-printable data to the output file. input and output must either be file objects or objects that mimic the file object interface. input will be read until input.readline() returns an empty string. quotetabs is a flag which controls whether to encode embedded spaces and tabs; when true it encodes such embedded whitespace, and when false it leaves them unencoded. Note that spaces and tabs appearing at the end of lines are always encoded, as per RFC 1521.

quopri.decodestring(s[, header])
: Like decode(), except that it accepts a source string and returns the corresponding decoded string.

quopri.encodestring(s[, quotetabs])
: Like encode(), except that it accepts a source string and returns the corresponding encoded string. quotetabs is optional (defaulting to 0), and is passed straight through to encode().

See also:

Module mimify General utilities for processing of MIME messages.

Module base64 Encode and decode MIME base64 data

18.16 uu — Encode and decode uuencode files

Source code: Lib/uu.py

This module encodes and decodes files in uuencode format, allowing arbitrary binary data to be transferred over ASCII-only connections. Wherever a file argument is expected, the methods accept a file-like object. For backwards compatibility, a string containing a pathname is also accepted, and the corresponding file will be opened for reading and writing; the pathname `'-'` is understood to mean the standard input or output. However, this interface is deprecated; it's better for the caller to open the file itself, and be sure that, when required, the mode is `'rb'` or `'wb'` on Windows.

This code was contributed by Lance Ellinghouse, and modified by Jack Jansen.

The uu module defines the following functions:

uu.encode(in_file, out_file[, name[, mode]])
: Uuencode file in_file into file out_file. The uuencoded file will have the header specifying name and mode as the defaults for the results of decoding the file. The default defaults are taken from in_file, or `'-'` and 0666 respectively.

uu.decode(in_file[, out_file[, mode[, quiet]]])
: This call decodes uuencoded file in_file placing the result on file out_file. If out_file is a pathname, mode is used to set the permission bits if the file must be created. Defaults for out_file and mode are taken from the uuencode header. However, if the file specified in the header already exists, a uu.Error is raised.

decode() may print a warning to standard error if the input was produced by an incorrect uuencoder and Python could recover from that error. Setting quiet to a true value silences this warning.

exception uu.Error
: Subclass of Exception, this can be raised by uu.decode() under various situations, such as described above, but also including a badly formatted header, or truncated input file.

See also:

Module binascii Support module containing ASCII-to-binary and binary-to-ASCII conversions.

CHAPTER

NINETEEN

STRUCTURED MARKUP PROCESSING TOOLS

Python supports a variety of modules to work with various forms of structured data markup. This includes modules to work with the Standard Generalized Markup Language (SGML) and the Hypertext Markup Language (HTML), and several interfaces for working with the Extensible Markup Language (XML).

It is important to note that modules in the xml package require that there be at least one SAX-compliant XML parser available. Starting with Python 2.3, the Expat parser is included with Python, so the xml.parsers.expat module will always be available. You may still want to be aware of the PyXML add-on package; that package provides an extended set of XML libraries for Python.

The documentation for the xml.dom and xml.sax packages are the definition of the Python bindings for the DOM and SAX interfaces.

19.1 HTMLParser — Simple HTML and XHTML parser

Note: The HTMLParser module has been renamed to html.parser in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

New in version 2.2.

Source code: Lib/HTMLParser.py

This module defines a class HTMLParser which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML. Unlike the parser in htmllib, this parser is not based on the SGML parser in sgmllib.

class HTMLParser.HTMLParser
: An HTMLParser instance is fed HTML data and calls handler methods when start tags, end tags, text, comments, and other markup elements are encountered. The user should subclass HTMLParser and override its methods to implement the desired behavior.

  The HTMLParser class is instantiated without arguments.

  Unlike the parser in htmllib, this parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.

An exception is defined as well:

exception HTMLParser.HTMLParseError
: HTMLParser is able to handle broken markup, but in some cases it might raise this exception when it encounters an error while parsing. This exception provides three attributes: msg is a brief message

explaining the error, lineno is the number of the line on which the broken construct was detected, and offset is the number of characters into the line at which the construct starts.

19.1.1 Example HTML Parser Application

As a basic example, below is a simple HTML parser that uses the HTMLParser class to print out start tags, end tags and data as they are encountered:

```
from HTMLParser import HTMLParser

# create a subclass and override the handler methods
class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print "Encountered a start tag:", tag

    def handle_endtag(self, tag):
        print "Encountered an end tag :", tag

    def handle_data(self, data):
        print "Encountered some data  :", data

# instantiate the parser and fed it some HTML
parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

The output will then be:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

19.1.2 HTMLParser Methods

HTMLParser instances have the following methods:

HTMLParser.feed(data)
: Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or close() is called. data can be either unicode or str, but passing unicode is advised.

HTMLParser.close()
: Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call the HTMLParser base class method close().

HTMLParser.reset()
Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

HTMLParser.getpos()
Return current line number and offset.

HTMLParser.get_starttag_text()
Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML "as deployed" or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

The following methods are called when data or markup elements are encountered and they are meant to be overridden in a subclass. The base class implementations do nothing (except for handle_startendtag()):

HTMLParser.handle_starttag(tag, attrs)
This method is called to handle the start of a tag (e.g.).

The tag argument is the name of the tag converted to lower case. The attrs argument is a list of (name, value) pairs containing the attributes found inside the tag's <> brackets. The name will be translated to lower case, and quotes in the value have been removed, and character and entity references have been replaced.

For instance, for the tag <A HREF="https://www.cwi.nl/">, this method would be called as handle_starttag('a', [('href', 'https://www.cwi.nl/')]).

Changed in version 2.6: All entity references from htmlentitydefs are now replaced in the attribute values.

HTMLParser.handle_endtag(tag)
This method is called to handle the end tag of an element (e.g.).

The tag argument is the name of the tag converted to lower case.

HTMLParser.handle_startendtag(tag, attrs)
Similar to handle_starttag(), but called when the parser encounters an XHTML-style empty tag (). This method may be overridden by subclasses which require this particular lexical information; the default implementation simply calls handle_starttag() and handle_endtag().

HTMLParser.handle_data(data)
This method is called to process arbitrary data (e.g. text nodes and the content of <script>...</script> and <style>...</style>).

HTMLParser.handle_entityref(name)
This method is called to process a named character reference of the form &name; (e.g. >), where name is a general entity reference (e.g. 'gt').

HTMLParser.handle_charref(name)
This method is called to process decimal and hexadecimal numeric character references of the form &#NNN; and &#xNNN;. For example, the decimal equivalent for > is >, whereas the hexadecimal is >; in this case the method will receive '62' or 'x3E'.

HTMLParser.handle_comment(data)
This method is called when a comment is encountered (e.g. <!--comment-->).

For example, the comment <!-- comment --> will cause this method to be called with the argument ' comment '.

The content of Internet Explorer conditional comments (condcoms) will also be sent to this method, so, for <!--[if IE 9]>IE9-specific content<![endif]-->, this method will receive '[if IE 9]>IE9-specific content<![endif]'.

HTMLParser.handle_decl(decl)
: This method is called to handle an HTML doctype declaration (e.g. <!DOCTYPE html>).

  The decl parameter will be the entire contents of the declaration inside the <!...> markup (e.g. 'DOCTYPE html').

HTMLParser.handle_pi(data)
: This method is called when a processing instruction is encountered. The data parameter will contain the entire processing instruction. For example, for the processing instruction <?proc color='red'>, this method would be called as handle_pi("proc color='red'").

  Note: The HTMLParser class uses the SGML syntactic rules for processing instructions. An XHTML processing instruction using the trailing '?' will cause the '?' to be included in data.

HTMLParser.unknown_decl(data)
: This method is called when an unrecognized declaration is read by the parser.

  The data parameter will be the entire contents of the declaration inside the <![...]> markup. It is sometimes useful to be overridden by a derived class.

19.1.3 Examples

The following class implements a parser that will be used to illustrate more examples:

```
from HTMLParser import HTMLParser
from htmlentitydefs import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print "Start tag:", tag
        for attr in attrs:
            print "     attr:", attr

    def handle_endtag(self, tag):
        print "End tag  :", tag

    def handle_data(self, data):
        print "Data     :", data

    def handle_comment(self, data):
        print "Comment  :", data

    def handle_entityref(self, name):
        c = unichr(name2codepoint[name])
        print "Named ent:", c

    def handle_charref(self, name):
        if name.startswith('x'):
            c = unichr(int(name[1:], 16))
        else:
            c = unichr(int(name))
        print "Num ent  :", c

    def handle_decl(self, data):
        print "Decl     :", data
```

(continues on next page)

(continued from previous page)

```
parser = MyHTMLParser()
```

Parsing a doctype:

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.
↪dtd"
```

Parsing an element with a few attributes and a title:

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
    attr: ('src', 'python-logo.png')
    attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

The content of script and style elements is returned as is, without further parsing:

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
    attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
    attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

Parsing comments:

```
>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  :  a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

Parsing named and numeric character references and converting them to the correct char (note: these 3 references are all equivalent to `'>'`):

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

Feeding incomplete chunks to feed() works, but handle_data() might be called more than once:

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
```

(continues on next page)

(continued from previous page)

```
...
Start tag: span
Data     : buff
Data     : ered
Data     : text
End tag  : span
```

Parsing invalid HTML (e.g. unquoted attributes) also works:

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
    attr: ('class', 'link')
    attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

19.2 sgmllib — Simple SGML parser

Deprecated since version 2.6: The sgmllib module has been removed in Python 3.

This module defines a class SGMLParser which serves as the basis for parsing text files formatted in SGML (Standard Generalized Mark-up Language). In fact, it does not provide a full SGML parser — it only parses SGML insofar as it is used by HTML, and the module only exists as a base for the htmllib module. Another HTML parser which supports XHTML and offers a somewhat different interface is available in the HTMLParser module.

class sgmllib.SGMLParser
: The SGMLParser class is instantiated without arguments. The parser is hardcoded to recognize the following constructs:

  - Opening and closing tags of the form <tag attr="value" ...> and </tag>, respectively.
  - Numeric character references of the form &#name;.
  - Entity references of the form &name;.
  - SGML comments of the form <!--text-->. Note that spaces, tabs, and newlines are allowed between the trailing > and the immediately preceding --.

A single exception is defined as well:

exception sgmllib.SGMLParseError
: Exception raised by the SGMLParser class when it encounters an error while parsing.

  New in version 2.1.

SGMLParser instances have the following methods:

SGMLParser.reset()
: Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

SGMLParser.setnomoretags()
: Stop processing tags. Treat all following input as literal input (CDATA). (This is only provided so the HTML tag <PLAINTEXT> can be implemented.)

SGMLParser.setliteral()
Enter literal mode (CDATA mode).

SGMLParser.feed(data)
Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or close() is called.

SGMLParser.close()
Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call close().

SGMLParser.get_starttag_text()
Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML "as deployed" or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

SGMLParser.handle_starttag(tag, method, attributes)
This method is called to handle start tags for which either a start_tag() or do_tag() method has been defined. The tag argument is the name of the tag converted to lower case, and the method argument is the bound method which should be used to support semantic interpretation of the start tag. The attributes argument is a list of (name, value) pairs containing the attributes found inside the tag's <> brackets.

The name has been translated to lower case. Double quotes and backslashes in the value have been interpreted, as well as known character references and known entity references terminated by a semicolon (normally, entity references can be terminated by any non-alphanumerical character, but this would break the very common case of <A HREF="url?spam=1&eggs=2"> when eggs is a valid entity name).

For instance, for the tag <A HREF="http://www.cwi.nl/">, this method would be called as unknown_starttag('a', [('href', 'http://www.cwi.nl/')]). The base implementation simply calls method with attributes as the only argument.

New in version 2.5: Handling of entity and character references within attribute values.

SGMLParser.handle_endtag(tag, method)
This method is called to handle endtags for which an end_tag() method has been defined. The tag argument is the name of the tag converted to lower case, and the method argument is the bound method which should be used to support semantic interpretation of the end tag. If no end_tag() method is defined for the closing element, this handler is not called. The base implementation simply calls method.

SGMLParser.handle_data(data)
This method is called to process arbitrary data. It is intended to be overridden by a derived class; the base class implementation does nothing.

SGMLParser.handle_charref(ref)
This method is called to process a character reference of the form &#ref;. The base implementation uses convert_charref() to convert the reference to a string. If that method returns a string, it is passed to handle_data(), otherwise unknown_charref(ref) is called to handle the error.

Changed in version 2.5: Use convert_charref() instead of hard-coding the conversion.

SGMLParser.convert_charref(ref)
Convert a character reference to a string, or None. ref is the reference passed in as a string. In the base implementation, ref must be a decimal number in the range 0–255. It converts the code point found using the convert_codepoint() method. If ref is invalid or out of range, this method returns None. This method is called by the default handle_charref() implementation and by the attribute value parser.

New in version 2.5.

SGMLParser.convert_codepoint(codepoint)
Convert a code point to a str value. Encodings can be handled here if appropriate, though the rest of sgmllib is oblivious on this matter.

New in version 2.5.

SGMLParser.handle_entityref(ref)
This method is called to process a general entity reference of the form &ref; where ref is a general entity reference. It converts ref by passing it to convert_entityref(). If a translation is returned, it calls the method handle_data() with the translation; otherwise, it calls the method unknown_entityref(ref). The default entitydefs defines translations for &, ', >, <, and ".

Changed in version 2.5: Use convert_entityref() instead of hard-coding the conversion.

SGMLParser.convert_entityref(ref)
Convert a named entity reference to a str value, or None. The resulting value will not be parsed. ref will be only the name of the entity. The default implementation looks for ref in the instance (or class) variable entitydefs which should be a mapping from entity names to corresponding translations. If no translation is available for ref, this method returns None. This method is called by the default handle_entityref() implementation and by the attribute value parser.

New in version 2.5.

SGMLParser.handle_comment(comment)
This method is called when a comment is encountered. The comment argument is a string containing the text between the <!-- and --> delimiters, but not the delimiters themselves. For example, the comment <!--text--> will cause this method to be called with the argument 'text'. The default method does nothing.

SGMLParser.handle_decl(data)
Method called when an SGML declaration is read by the parser. In practice, the DOCTYPE declaration is the only thing observed in HTML, but the parser does not discriminate among different (or broken) declarations. Internal subsets in a DOCTYPE declaration are not supported. The data parameter will be the entire contents of the declaration inside the <!...> markup. The default implementation does nothing.

SGMLParser.report_unbalanced(tag)
This method is called when an end tag is found which does not correspond to any open element.

SGMLParser.unknown_starttag(tag, attributes)
This method is called to process an unknown start tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

SGMLParser.unknown_endtag(tag)
This method is called to process an unknown end tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

SGMLParser.unknown_charref(ref)
This method is called to process unresolvable numeric character references. Refer to handle_charref() to determine what is handled by default. It is intended to be overridden by a derived class; the base class implementation does nothing.

SGMLParser.unknown_entityref(ref)
This method is called to process an unknown entity reference. It is intended to be overridden by a derived class; the base class implementation does nothing.

Apart from overriding or extending the methods listed above, derived classes may also define methods of the following form to define processing of specific tags. Tag names in the input stream are case independent; the tag occurring in method names must be in lower case:

SGMLParser.start_tag(attributes)
: This method is called to process an opening tag tag. It has preference over do_tag(). The attributes argument has the same meaning as described for handle_starttag() above.

SGMLParser.do_tag(attributes)
: This method is called to process an opening tag tag for which no start_tag() method is defined. The attributes argument has the same meaning as described for handle_starttag() above.

SGMLParser.end_tag()
: This method is called to process a closing tag tag.

Note that the parser maintains a stack of open elements for which no end tag has been found yet. Only tags processed by start_tag() are pushed on this stack. Definition of an end_tag() method is optional for these tags. For tags processed by do_tag() or by unknown_tag(), no end_tag() method must be defined; if defined, it will not be used. If both start_tag() and do_tag() methods exist for a tag, the start_tag() method takes precedence.

19.3 htmllib — A parser for HTML documents

Deprecated since version 2.6: The htmllib module has been removed in Python 3. Use HTMLParser instead in Python 2, and the equivalent, html.parser, in Python 3.

This module defines a class which can serve as a base for parsing text files formatted in the HyperText Mark-up Language (HTML). The class is not directly concerned with I/O — it must be provided with input in string form via a method, and makes calls to methods of a "formatter" object in order to produce output. The HTMLParser class is designed to be used as a base class for other classes in order to add functionality, and allows most of its methods to be extended or overridden. In turn, this class is derived from and extends the SGMLParser class defined in module sgmllib. The HTMLParser implementation supports the HTML 2.0 language as described in RFC 1866. Two implementations of formatter objects are provided in the formatter module; refer to the documentation for that module for information on the formatter interface.

The following is a summary of the interface defined by sgmllib.SGMLParser:

- The interface to feed data to an instance is through the feed() method, which takes a string argument. This can be called with as little or as much text at a time as desired; p.feed(a); p.feed(b) has the same effect as p.feed(a+b). When the data contains complete HTML markup constructs, these are processed immediately; incomplete constructs are saved in a buffer. To force processing of all unprocessed data, call the close() method.

  For example, to parse the entire contents of a file, use:

  ```
  parser.feed(open('myfile.html').read())
  parser.close()
  ```

- The interface to define semantics for HTML tags is very simple: derive a class and define methods called start_tag(), end_tag(), or do_tag(). The parser will call these at appropriate moments: start_tag() or do_tag() is called when an opening tag of the form <tag ...> is encountered; end_tag() is called when a closing tag of the form <tag> is encountered. If an opening tag requires a corresponding closing tag, like <H1> ... </H1>, the class should define the start_tag() method; if a tag requires no closing tag, like <P>, the class should define the do_tag() method.

The module defines a parser class and an exception:

class htmllib.HTMLParser(formatter)
: This is the basic HTML parser class. It supports all entity names required by the XHTML 1.0 Recommendation (https://www.w3.org/TR/xhtml1). It also defines handlers for all HTML 2.0 and many HTML 3.0 and 3.2 elements.

exception htmllib.HTMLParseError
Exception raised by the HTMLParser class when it encounters an error while parsing.

New in version 2.4.

See also:

Module formatter Interface definition for transforming an abstract flow of formatting events into specific output events on writer objects.

Module HTMLParser Alternate HTML parser that offers a slightly lower-level view of the input, but is designed to work with XHTML, and does not implement some of the SGML syntax not used in "HTML as deployed" and which isn't legal for XHTML.

Module htmlentitydefs Definition of replacement text for XHTML 1.0 entities.

Module sgmllib Base class for HTMLParser.

19.3.1 HTMLParser Objects

In addition to tag methods, the HTMLParser class provides some additional methods and instance variables for use within tag methods.

HTMLParser.formatter
This is the formatter instance associated with the parser.

HTMLParser.nofill
Boolean flag which should be true when whitespace should not be collapsed, or false when it should be. In general, this should only be true when character data is to be treated as "preformatted" text, as within a <PRE> element. The default value is false. This affects the operation of handle_data() and save_end().

HTMLParser.anchor_bgn(href, name, type)
This method is called at the start of an anchor region. The arguments correspond to the attributes of the <A> tag with the same names. The default implementation maintains a list of hyperlinks (defined by the HREF attribute for <A> tags) within the document. The list of hyperlinks is available as the data attribute anchorlist.

HTMLParser.anchor_end()
This method is called at the end of an anchor region. The default implementation adds a textual footnote marker using an index into the list of hyperlinks created by anchor_bgn().

HTMLParser.handle_image(source, alt[, ismap[, align[, width[, height]]]])
This method is called to handle images. The default implementation simply passes the alt value to the handle_data() method.

HTMLParser.save_bgn()
Begins saving character data in a buffer instead of sending it to the formatter object. Retrieve the stored data via save_end(). Use of the save_bgn() / save_end() pair may not be nested.

HTMLParser.save_end()
Ends buffering character data and returns all data saved since the preceding call to save_bgn(). If the nofill flag is false, whitespace is collapsed to single spaces. A call to this method without a preceding call to save_bgn() will raise a TypeError exception.

19.4 htmlentitydefs — Definitions of HTML general entities

Note: The htmlentitydefs module has been renamed to html.entities in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/htmlentitydefs.py

This module defines three dictionaries, name2codepoint, codepoint2name, and entitydefs. entitydefs is used by the htmllib module to provide the entitydefs attribute of the HTMLParser class. The definition provided here contains all the entities defined by XHTML 1.0 that can be handled using simple textual substitution in the Latin-1 character set (ISO-8859-1).

htmlentitydefs.entitydefs
: A dictionary mapping XHTML 1.0 entity definitions to their replacement text in ISO Latin-1.

htmlentitydefs.name2codepoint
: A dictionary that maps HTML entity names to the Unicode code points.

  New in version 2.3.

htmlentitydefs.codepoint2name
: A dictionary that maps Unicode code points to HTML entity names.

  New in version 2.3.

19.5 XML Processing Modules

Python's interfaces for processing XML are grouped in the xml package.

Warning: The XML modules are not secure against erroneous or maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

It is important to note that modules in the xml package require that there be at least one SAX-compliant XML parser available. The Expat parser is included with Python, so the xml.parsers.expat module will always be available.

The documentation for the xml.dom and xml.sax packages are the definition of the Python bindings for the DOM and SAX interfaces.

The XML handling submodules are:

- xml.etree.ElementTree: the ElementTree API, a simple and lightweight XML processor
- xml.dom: the DOM API definition
- xml.dom.minidom: a minimal DOM implementation
- xml.dom.pulldom: support for building partial DOM trees
- xml.sax: SAX2 base classes and convenience functions
- xml.parsers.expat: the Expat parser binding

19.6 XML vulnerabilities

The XML processing modules are not secure against maliciously constructed data. An attacker can abuse vulnerabilities for e.g. denial of service attacks, to access local files, to generate network connections to other machines, or to or circumvent firewalls. The attacks on XML abuse unfamiliar features like inline DTD (document type definition) with entities.

The following table gives an overview of the known attacks and if the various modules are vulnerable to them.

kind	sax	etree	minidom	pulldom	xmlrpc
billion laughs	Vulnerable	Vulnerable	Vulnerable	Vulnerable	Vulnerable
quadratic blowup	Vulnerable	Vulnerable	Vulnerable	Vulnerable	Vulnerable
external entity expansion	Vulnerable	Safe (1)	Safe (2)	Vulnerable	Safe (3)
DTD retrieval	Vulnerable	Safe	Safe	Vulnerable	Safe
decompression bomb	Safe	Safe	Safe	Safe	Vulnerable

1. xml.etree.ElementTree doesn't expand external entities and raises a ParserError when an entity occurs.
2. xml.dom.minidom doesn't expand external entities and simply returns the unexpanded entity verbatim.
3. xmlrpclib doesn't expand external entities and omits them.

billion laughs / exponential entity expansion The Billion Laughs attack – also known as exponential entity expansion – uses multiple levels of nested entities. Each entity refers to another entity several times, the final entity definition contains a small string. Eventually the small string is expanded to several gigabytes. The exponential expansion consumes lots of CPU time, too.

quadratic blowup entity expansion A quadratic blowup attack is similar to a Billion Laughs attack; it abuses entity expansion, too. Instead of nested entities it repeats one large entity with a couple of thousand chars over and over again. The attack isn't as efficient as the exponential case but it avoids triggering countermeasures of parsers against heavily nested entities.

external entity expansion Entity declarations can contain more than just text for replacement. They can also point to external resources by public identifiers or system identifiers. System identifiers are standard URIs or can refer to local files. The XML parser retrieves the resource with e.g. HTTP or FTP requests and embeds the content into the XML document.

DTD retrieval Some XML libraries like Python's xml.dom.pulldom retrieve document type definitions from remote or local locations. The feature has similar implications as the external entity expansion issue.

decompression bomb The issue of decompression bombs (aka ZIP bomb) apply to all XML libraries that can parse compressed XML stream like gzipped HTTP streams or LZMA-ed files. For an attacker it can reduce the amount of transmitted data by three magnitudes or more.

The documentation of defusedxml on PyPI has further information about all known attack vectors with examples and references.

19.6.1 defused packages

These external packages are recommended for any code that parses untrusted XML data.

defusedxml is a pure Python package with modified subclasses of all stdlib XML parsers that prevent any potentially malicious operation. The package also ships with example exploits and extended documentation on more XML exploits like xpath injection.

defusedexpat provides a modified libexpat and patched replacement pyexpat extension module with countermeasures against entity expansion DoS attacks. Defusedexpat still allows a sane and configurable amount of entity expansions. The modifications will be merged into future releases of Python.

The workarounds and modifications are not included in patch releases as they break backward compatibility. After all inline DTD and entity expansion are well-defined XML features.

19.7 xml.etree.ElementTree — The ElementTree XML API

New in version 2.5.

Source code: Lib/xml/etree/ElementTree.py

The Element type is a flexible container object, designed to store hierarchical data structures in memory. The type can be described as a cross between a list and a dictionary.

> Warning: The xml.etree.ElementTree module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

Each element has a number of properties associated with it:

- a tag which is a string identifying what kind of data this element represents (the element type, in other words).
- a number of attributes, stored in a Python dictionary.
- a text string.
- an optional tail string.
- a number of child elements, stored in a Python sequence

To create an element instance, use the Element constructor or the SubElement() factory function.

The ElementTree class can be used to wrap an element structure, and convert it from and to XML.

A C implementation of this API is available as xml.etree.cElementTree.

See http://effbot.org/zone/element-index.htm for tutorials and links to other docs. Fredrik Lundh's page is also the location of the development version of the xml.etree.ElementTree.

Changed in version 2.7: The ElementTree API is updated to 1.3. For more information, see Introducing ElementTree 1.3.

19.7.1 Tutorial

This is a short tutorial for using xml.etree.ElementTree (ET in short). The goal is to demonstrate some of the building blocks and basic concepts of the module.

XML tree and elements

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ET has two classes for this purpose - ElementTree represents the whole XML document as a tree, and Element represents a single node in this tree. Interactions with the whole document (reading and writing to/from

files) are usually done on the ElementTree level. Interactions with a single XML element and its sub-elements are done on the Element level.

Parsing XML

We'll be using the following XML document as the sample data for this section:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

We have a number of ways to import the data. Reading the file from disk:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Reading the data from a string:

```
root = ET.fromstring(country_data_as_string)
```

fromstring() parses XML from a string directly into an Element, which is the root element of the parsed tree. Other parsing functions may create an ElementTree. Check the documentation to be sure.

As an Element, root has a tag and a dictionary of attributes:

```
>>> root.tag
'data'
>>> root.attrib
{}
```

It also has children nodes over which we can iterate:

```
>>> for child in root:
...     print child.tag, child.attrib
...
country {'name': 'Liechtenstein'}
```

(continues on next page)

(continued from previous page)

```
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Children are nested, and we can access specific child nodes by index:

```
>>> root[0][1].text
'2008'
```

Finding interesting elements

Element has some useful methods that help iterate recursively over all the sub-tree below it (its children, their children, and so on). For example, Element.iter():

```
>>> for neighbor in root.iter('neighbor'):
...     print neighbor.attrib
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

Element.findall() finds only elements with a tag which are direct children of the current element. Element.find() finds the first child with a particular tag, and Element.text accesses the element's text content. Element.get() accesses the element's attributes:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print name, rank
...
Liechtenstein 1
Singapore 4
Panama 68
```

More sophisticated specification of which elements to look for is possible by using XPath.

Modifying an XML File

ElementTree provides a simple way to build XML documents and write them to files. The ElementTree.write() method serves this purpose.

Once created, an Element object may be manipulated by directly changing its fields (such as Element.text), adding and modifying attributes (Element.set() method), as well as adding new children (for example with Element.append()).

Let's say we want to add one to each country's rank, and add an updated attribute to the rank element:

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

Our XML now looks like this:

```
<?xml version="1.0"?>
<data>
   <country name="Liechtenstein">
      <rank updated="yes">2</rank>
      <year>2008</year>
      <gdppc>141100</gdppc>
      <neighbor name="Austria" direction="E"/>
      <neighbor name="Switzerland" direction="W"/>
   </country>
   <country name="Singapore">
      <rank updated="yes">5</rank>
      <year>2011</year>
      <gdppc>59900</gdppc>
      <neighbor name="Malaysia" direction="N"/>
   </country>
   <country name="Panama">
      <rank updated="yes">69</rank>
      <year>2011</year>
      <gdppc>13600</gdppc>
      <neighbor name="Costa Rica" direction="W"/>
      <neighbor name="Colombia" direction="E"/>
   </country>
</data>
```

We can remove elements using Element.remove(). Let's say we want to remove all countries with a rank higher than 50:

```
>>> for country in root.findall('country'):
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

Our XML now looks like this:

```
<?xml version="1.0"?>
<data>
   <country name="Liechtenstein">
      <rank updated="yes">2</rank>
      <year>2008</year>
      <gdppc>141100</gdppc>
      <neighbor name="Austria" direction="E"/>
      <neighbor name="Switzerland" direction="W"/>
   </country>
   <country name="Singapore">
      <rank updated="yes">5</rank>
      <year>2011</year>
      <gdppc>59900</gdppc>
      <neighbor name="Malaysia" direction="N"/>
   </country>
</data>
```

Building XML documents

The SubElement() function also provides a convenient way to create new sub-elements for a given element:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

Parsing XML with Namespaces

If the XML input has namespaces, tags and attributes with prefixes in the form prefix:sometag get expanded to {uri}sometag where the prefix is replaced by the full URI. Also, if there is a default namespace, that full URI gets prepended to all of the non-prefixed tags.

Here is an XML example that incorporates two namespaces, one with the prefix "fictional" and the other serving as the default namespace:

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
    </actor>
</actors>
```

One way to search and explore this XML example is to manually add the URI to every tag or attribute in the xpath of a find() or findall():

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print name.text
    for char in actor.findall('{http://characters.example.com}character'):
        print ' |-->', char.text
```

A better way to search the namespaced XML example is to create a dictionary with your own prefixes and use those in the search functions:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print name.text
    for char in actor.findall('role:character', ns):
        print ' |-->', char.text
```

These two approaches both output:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

Additional resources

See http://effbot.org/zone/element-index.htm for tutorials and links to other docs.

19.7.2 XPath support

This module provides limited support for XPath expressions for locating elements in a tree. The goal is to support a small subset of the abbreviated syntax; a full XPath engine is outside the scope of the module.

Example

Here's an example that demonstrates some of the XPath capabilities of the module. We'll be using the countrydata XML document from the Parsing XML section:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

Supported XPath syntax

Syntax	Meaning
tag	Selects all child elements with the given tag. For example, spam selects all child elements named spam, and spam/egg selects all grandchildren named egg in all children named spam.
*	Selects all child elements. For example, */egg selects all grandchildren named egg.
.	Selects the current node. This is mostly useful at the beginning of the path, to indicate that it's a relative path.
//	Selects all subelements, on all levels beneath the current element. For example, .//egg selects all egg elements in the entire tree.
..	Selects the parent element.
[@attrib]	Selects all elements that have the given attribute.
[@attrib='value']	Selects all elements for which the given attribute has the given value. The value cannot contain quotes.
[tag]	Selects all elements that have a child named tag. Only immediate children are supported.
[tag='text']	Selects all elements that have a child named tag whose complete text content, including descendants, equals the given text.
[position]	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression last() (for the last position), or a position relative to the last position (e.g. last()-1).

Predicates (expressions within square brackets) must be preceded by a tag name, an asterisk, or another predicate. position predicates must be preceded by a tag name.

19.7.3 Reference

Functions

xml.etree.ElementTree.Comment(text=None)
: Comment element factory. This factory function creates a special element that will be serialized as an XML comment by the standard serializer. The comment string can be either a bytestring or a Unicode string. text is a string containing the comment string. Returns an element instance representing a comment.

xml.etree.ElementTree.dump(elem)
: Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

  The exact output format is implementation dependent. In this version, it's written as an ordinary XML file.

  elem is an element tree or an individual element.

xml.etree.ElementTree.fromstring(text)
: Parses an XML section from a string constant. Same as XML(). text is a string containing XML data. Returns an Element instance.

xml.etree.ElementTree.fromstringlist(sequence, parser=None)
: Parses an XML document from a sequence of string fragments. sequence is a list or other sequence containing XML data fragments. parser is an optional parser instance. If not given, the standard XMLParser parser is used. Returns an Element instance.

  New in version 2.7.

xml.etree.ElementTree.iselement(element)
Checks if an object appears to be a valid element object. element is an element instance. Returns a true value if this is an element object.

xml.etree.ElementTree.iterparse(source, events=None, parser=None)
Parses an XML section into an element tree incrementally, and reports what's going on to the user. source is a filename or file object containing XML data. events is a list of events to report back. If omitted, only "end" events are reported. parser is an optional parser instance. If not given, the standard XMLParser parser is used. parser is not supported by cElementTree. Returns an iterator providing (event, elem) pairs.

Note: iterparse() only guarantees that it has seen the ">" character of a starting tag when it emits a "start" event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.

If you need a fully populated element, look for "end" events instead.

---

xml.etree.ElementTree.parse(source, parser=None)
Parses an XML section into an element tree. source is a filename or file object containing XML data. parser is an optional parser instance. If not given, the standard XMLParser parser is used. Returns an ElementTree instance.

xml.etree.ElementTree.ProcessingInstruction(target, text=None)
PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. target is a string containing the PI target. text is a string containing the PI contents, if given. Returns an element instance, representing a processing instruction.

xml.etree.ElementTree.register_namespace(prefix, uri)
Registers a namespace prefix. The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed. prefix is a namespace prefix. uri is a namespace uri. Tags and attributes in this namespace will be serialized with the given prefix, if at all possible.

New in version 2.7.

xml.etree.ElementTree.SubElement(parent, tag, attrib={}, **extra)
Subelement factory. This function creates an element instance, and appends it to an existing element.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. parent is the parent element. tag is the subelement name. attrib is an optional dictionary, containing element attributes. extra contains additional attributes, given as keyword arguments. Returns an element instance.

xml.etree.ElementTree.tostring(element, encoding="us-ascii", method="xml")
Generates a string representation of an XML element, including all subelements. element is an Element instance. encoding[1] is the output encoding (default is US-ASCII). method is either "xml", "html" or "text" (default is "xml"). Returns an encoded string containing the XML data.

xml.etree.ElementTree.tostringlist(element, encoding="us-ascii", method="xml")
Generates a string representation of an XML element, including all subelements. element is an Element instance. encoding[1] is the output encoding (default is US-ASCII). method is either "xml", "html" or "text" (default is "xml"). Returns a list of encoded strings containing the XML data. It does not guarantee any specific sequence, except that "".join(tostringlist(element)) == tostring(element).

New in version 2.7.

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

xml.etree.ElementTree.XML(text, parser=None)
Parses an XML section from a string constant. This function can be used to embed "XML literals" in Python code. text is a string containing XML data. parser is an optional parser instance. If not given, the standard XMLParser parser is used. Returns an Element instance.

xml.etree.ElementTree.XMLID(text, parser=None)
Parses an XML section from a string constant, and also returns a dictionary which maps from element id:s to elements. text is a string containing XML data. parser is an optional parser instance. If not given, the standard XMLParser parser is used. Returns a tuple containing an Element instance and a dictionary.

Element Objects

class xml.etree.ElementTree.Element(tag, attrib={}, **extra)
Element class. This class defines the Element interface, and provides a reference implementation of this interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. tag is the element name. attrib is an optional dictionary, containing element attributes. extra contains additional attributes, given as keyword arguments.

tag
A string identifying what kind of data this element represents (the element type, in other words).

text
tail
These attributes can be used to hold additional data associated with the element. Their values are usually strings but may be any application-specific object. If the element is created from an XML file, the text attribute holds either the text between the element's start tag and its first child or end tag, or None, and the tail attribute holds either the text between the element's end tag and the next tag, or None. For the XML data

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

the a element has None for both text and tail attributes, the b element has text "1" and tail "4", the c element has text "2" and tail None, and the d element has text None and tail "3".

To collect the inner text of an element, see itertext(), for example "".join(element.itertext()).

Applications may store arbitrary objects in these attributes.

attrib
A dictionary containing the element's attributes. Note that while the attrib value is always a real mutable Python dictionary, an ElementTree implementation may choose to use another internal representation, and create the dictionary only if someone asks for it. To take advantage of such implementations, use the dictionary methods below whenever possible.

The following dictionary-like methods work on the element attributes.

clear()
Resets an element. This function removes all subelements, clears all attributes, and sets the text and tail attributes to None.

get(key, default=None)
Gets the element attribute named key.

Returns the attribute value, or default if the attribute was not found.

items()
: Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

keys()
: Returns the elements attribute names as a list. The names are returned in an arbitrary order.

set(key, value)
: Set the attribute key on the element to value.

The following methods work on the element's children (subelements).

append(subelement)
: Adds the element subelement to the end of this elements internal list of subelements.

extend(subelements)
: Appends subelements from a sequence object with zero or more elements. Raises AssertionError if a subelement is not a valid object.

  New in version 2.7.

find(match)
: Finds the first subelement matching match. match may be a tag name or path. Returns an element instance or None.

findall(match)
: Finds all matching subelements, by tag name or path. Returns a list containing all matching elements in document order.

findtext(match, default=None)
: Finds text for the first subelement matching match. match may be a tag name or path. Returns the text content of the first matching element, or default if no element was found. Note that if the matching element has no text content an empty string is returned.

getchildren()
: Deprecated since version 2.7: Use list(elem) or iteration.

getiterator(tag=None)
: Deprecated since version 2.7: Use method Element.iter() instead.

insert(index, element)
: Inserts a subelement at the given position in this element.

iter(tag=None)
: Creates a tree iterator with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If tag is not None or `'*'`, only elements whose tag equals tag are returned from the iterator. If the tree structure is modified during iteration, the result is undefined.

  New in version 2.7.

iterfind(match)
: Finds all matching subelements, by tag name or path. Returns an iterable yielding all matching elements in document order.

  New in version 2.7.

itertext()
: Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text.

  New in version 2.7.

makeelement(tag, attrib)
: Creates a new element object of the same type as this element. Do not call this method, use the SubElement() factory function instead.

remove(subelement)
: Removes subelement from the element. Unlike the find* methods this method compares elements based on the instance identity, not on tag value or contents.

Element objects also support the following sequence type methods for working with subelements: __delitem__(), __getitem__(), __setitem__(), __len__().

Caution: Elements with no subelements will test as False. This behavior will change in future versions. Use specific len(elem) or elem is None test instead.

```
element = root.find('foo')

if not element:  # careful!
    print "element not found, or element has no subelements"

if element is None:
    print "element not found"
```

ElementTree Objects

class xml.etree.ElementTree.ElementTree(element=None, file=None)
: ElementTree wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

element is the root element. The tree is initialized with the contents of the XML file if given.

_setroot(element)
: Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. element is an element instance.

find(match)
: Same as Element.find(), starting at the root of the tree.

findall(match)
: Same as Element.findall(), starting at the root of the tree.

findtext(match, default=None)
: Same as Element.findtext(), starting at the root of the tree.

getiterator(tag=None)
: Deprecated since version 2.7: Use method ElementTree.iter() instead.

getroot()
: Returns the root element for this tree.

iter(tag=None)
: Creates and returns a tree iterator for the root element. The iterator loops over all elements in this tree, in section order. tag is the tag to look for (default is to return all elements).

iterfind(match)
: Finds all matching subelements, by tag name or path. Same as getroot().iterfind(match). Returns an iterable yielding all matching elements in document order.

  New in version 2.7.

parse(source, parser=None)
Loads an external XML section into this element tree. source is a file name or file object. parser is an optional parser instance. If not given, the standard XMLParser parser is used. Returns the section root element.

write(file, encoding="us-ascii", xml_declaration=None, default_namespace=None, method="xml")
Writes the element tree to a file, as XML. file is a file name, or a file object opened for writing. encoding[1] is the output encoding (default is US-ASCII). xml_declaration controls if an XML declaration should be added to the file. Use False for never, True for always, None for only if not US-ASCII or UTF-8 (default is None). default_namespace sets the default XML namespace (for "xmlns"). method is either "xml", "html" or "text" (default is "xml"). Returns an encoded string.

This is the XML file that is going to be manipulated:

```
<html>
  <head>
    <title>Example page</title>
  </head>
  <body>
    <p>Moved to <a href="http://example.org/">example.org</a>
    or <a href="http://example.com/">example.com</a>.</p>
  </body>
</html>
```

Example of changing the attribute "target" of every link in first paragraph:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")     # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))   # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:             # Iterates through all found links
...     i.attrib["target"] = "blank"
...
>>> tree.write("output.xhtml")
```

QName Objects

class xml.etree.ElementTree.QName(text_or_uri, tag=None)
QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. text_or_uri is a string containing the QName value, in the form {uri}local, or, if the tag argument is given, the URI part of a QName. If tag is given, the first argument is interpreted as a URI, and this argument is interpreted as a local name. QName instances are opaque.

TreeBuilder Objects

class xml.etree.ElementTree.TreeBuilder(element_factory=None)
Generic element structure builder. This builder converts a sequence of start, data, and end method

calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format. The element_factory is called to create new Element instances when given.

close()
: Flushes the builder buffers, and returns the toplevel document element. Returns an Element instance.

data(data)
: Adds text to the current element. data is a string. This should be either a bytestring, or a Unicode string.

end(tag)
: Closes the current element. tag is the element name. Returns the closed element.

start(tag, attrs)
: Opens a new element. tag is the element name. attrs is a dictionary containing element attributes. Returns the opened element.

In addition, a custom TreeBuilder object can provide the following method:

doctype(name, pubid, system)
: Handles a doctype declaration. name is the doctype name. pubid is the public identifier. system is the system identifier. This method does not exist on the default TreeBuilder class.

  New in version 2.7.

XMLParser Objects

class xml.etree.ElementTree.XMLParser(html=0, target=None, encoding=None)
: Element structure builder for XML source data, based on the expat parser. html are predefined HTML entities. This flag is not supported by the current implementation. target is the target object. If omitted, the builder uses an instance of the standard TreeBuilder class. encoding[1] is optional. If given, the value overrides the encoding specified in the XML file.

  close()
  : Finishes feeding data to the parser. Returns an element structure.

  doctype(name, pubid, system)
  : Deprecated since version 2.7: Define the TreeBuilder.doctype() method on a custom TreeBuilder target.

  feed(data)
  : Feeds data to the parser. data is encoded data.

XMLParser.feed() calls target's start() method for each opening tag, its end() method for each closing tag, and data is processed by method data(). XMLParser.close() calls target's method close(). XMLParser can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing tag.
```

(continues on next page)

(continued from previous page)

```
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything with data.
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

19.8 xml.dom — The Document Object Model API

New in version 2.0.

The Document Object Model, or "DOM," is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents. A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch. It then gives access to the structure through a set of objects which provided well-known interfaces.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another. If you are looking at a text node, you have no access to a containing element. When you write a SAX application, you need to keep track of your program's position in the document somewhere in your own code. SAX does not do it for you. Also, if you need to look ahead in the XML document, you are just out of luck.

Some applications are simply impossible in an event driven model with no access to a tree. Of course you could build some sort of tree yourself in SAX events, but the DOM allows you to avoid writing that code. The DOM is a standard tree representation for XML data.

The Document Object Model is being defined by the W3C in stages, or "levels" in their terminology. The Python mapping of the API is substantially based on the DOM Level 2 recommendation.

DOM applications typically start by parsing some XML into a DOM. How this is accomplished is not covered at all by DOM Level 1, and Level 2 provides only limited improvements: There is a DOMImplementation object class which provides access to Document creation methods, but no way to access an XML reader/parser/Document builder in an implementation-independent way. There is also no well-defined way to access these methods without an existing Document object. In Python, each DOM implementation will provide a function getDOMImplementation(). DOM Level 3 adds a Load/Store specification, which defines an interface to the reader, but this is not yet available in the Python standard library.

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification; this portion of the reference

manual describes the interpretation of the specification in Python.

The specification provided by the W3C defines the DOM API for Java, ECMAScript, and OMG IDL. The Python mapping defined here is based in large part on the IDL version of the specification, but strict compliance is not required (though implementations are free to support the strict mapping from IDL). See section Conformance for a detailed discussion of mapping requirements.

See also:

Document Object Model (DOM) Level 2 Specification The W3C recommendation upon which the Python DOM API is based.

Document Object Model (DOM) Level 1 Specification The W3C recommendation for the DOM supported by xml.dom.minidom.

Python Language Mapping Specification This specifies the mapping from OMG IDL to Python.

19.8.1 Module Contents

The xml.dom contains the following functions:

xml.dom.registerDOMImplementation(name, factory)
Register the factory function with the name name. The factory function should return an object which implements the DOMImplementation interface. The factory function can return the same object every time, or a new one for each call, as appropriate for the specific implementation (e.g. if that implementation supports some customization).

xml.dom.getDOMImplementation([name[, features]])
Return a suitable DOM implementation. The name is either well-known, the module name of a DOM implementation, or None. If it is not None, imports the corresponding module and returns a DOMImplementation object if the import succeeds. If no name is given, and if the environment variable PYTHON_DOM is set, this variable is used to find the implementation.

If name is not given, this examines the available implementations to find one with the required feature set. If no implementation can be found, raise an ImportError. The features list must be a sequence of (feature, version) pairs which are passed to the hasFeature() method on available DOMImplementation objects.

Some convenience constants are also provided:

xml.dom.EMPTY_NAMESPACE
The value used to indicate that no namespace is associated with a node in the DOM. This is typically found as the namespaceURI of a node, or used as the namespaceURI parameter to a namespaces-specific method.

New in version 2.2.

xml.dom.XML_NAMESPACE
The namespace URI associated with the reserved prefix xml, as defined by Namespaces in XML (section 4).

New in version 2.2.

xml.dom.XMLNS_NAMESPACE
The namespace URI for namespace declarations, as defined by Document Object Model (DOM) Level 2 Core Specification (section 1.1.8).

New in version 2.2.

xml.dom.XHTML_NAMESPACE
: The URI of the XHTML namespace as defined by XHTML 1.0: The Extensible HyperText Markup Language (section 3.1.1).

  New in version 2.2.

In addition, xml.dom contains a base Node class and the DOM exception classes. The Node class provided by this module does not implement any of the methods or attributes defined by the DOM specification; concrete DOM implementations must provide those. The Node class provided as part of this module does provide the constants used for the nodeType attribute on concrete Node objects; they are located within the class rather than at the module level to conform with the DOM specifications.

19.8.2 Objects in the DOM

The definitive documentation for the DOM is the DOM specification from the W3C.

Note that DOM attributes may also be manipulated as nodes instead of as simple strings. It is fairly rare that you must do this, however, so this usage is not yet documented.

Interface	Section	Purpose
DOMImplementation	DOMImplementation Objects	Interface to the underlying implementation.
Node	Node Objects	Base interface for most objects in a document.
NodeList	NodeList Objects	Interface for a sequence of nodes.
DocumentType	DocumentType Objects	Information about the declarations needed to process a document.
Document	Document Objects	Object which represents an entire document.
Element	Element Objects	Element nodes in the document hierarchy.
Attr	Attr Objects	Attribute value nodes on element nodes.
Comment	Comment Objects	Representation of comments in the source document.
Text	Text and CDATASection Objects	Nodes containing textual content from the document.
ProcessingInstruction	ProcessingInstruction Objects	Processing instruction representation.

An additional section describes the exceptions defined for working with the DOM in Python.

DOMImplementation Objects

The DOMImplementation interface provides a way for applications to determine the availability of particular features in the DOM they are using. DOM Level 2 added the ability to create new Document and DocumentType objects using the DOMImplementation as well.

DOMImplementation.hasFeature(feature, version)
: Return true if the feature identified by the pair of strings feature and version is implemented.

DOMImplementation.createDocument(namespaceUri, qualifiedName, doctype)
: Return a new Document object (the root of the DOM), with a child Element object having the given namespaceUri and qualifiedName. The doctype must be a DocumentType object created by createDocumentType(), or None. In the Python DOM API, the first two arguments can also be None in order to indicate that no Element child is to be created.

DOMImplementation.createDocumentType(qualifiedName, publicId, systemId)
: Return a new DocumentType object that encapsulates the given qualifiedName, publicId, and systemId strings, representing the information contained in an XML document type declaration.

Node Objects

All of the components of an XML document are subclasses of Node.

Node.nodeType
: An integer representing the node type. Symbolic constants for the types are on the Node object: ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE, ENTITY_NODE, PROCESSING_INSTRUCTION_NODE, COMMENT_NODE, DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE. This is a read-only attribute.

Node.parentNode
: The parent of the current node, or None for the document node. The value is always a Node object or None. For Element nodes, this will be the parent element, except for the root element, in which case it will be the Document object. For Attr nodes, this is always None. This is a read-only attribute.

Node.attributes
: A NamedNodeMap of attribute objects. Only elements have actual values for this; others provide None for this attribute. This is a read-only attribute.

Node.previousSibling
: The node that immediately precedes this one with the same parent. For instance the element with an end-tag that comes just before the self element's start-tag. Of course, XML documents are made up of more than just elements so the previous sibling could be text, a comment, or something else. If this node is the first child of the parent, this attribute will be None. This is a read-only attribute.

Node.nextSibling
: The node that immediately follows this one with the same parent. See also previousSibling. If this is the last child of the parent, this attribute will be None. This is a read-only attribute.

Node.childNodes
: A list of nodes contained within this node. This is a read-only attribute.

Node.firstChild
: The first child of the node, if there are any, or None. This is a read-only attribute.

Node.lastChild
: The last child of the node, if there are any, or None. This is a read-only attribute.

Node.localName
: The part of the tagName following the colon if there is one, else the entire tagName. The value is a string.

Node.prefix
: The part of the tagName preceding the colon if there is one, else the empty string. The value is a string, or None.

Node.namespaceURI
: The namespace associated with the element name. This will be a string or None. This is a read-only attribute.

Node.nodeName
: This has a different meaning for each node type; see the DOM specification for details. You can always get the information you would get here from another property such as the tagName property for elements or the name property for attributes. For all node types, the value of this attribute will be either a string or None. This is a read-only attribute.

Node.nodeValue
: This has a different meaning for each node type; see the DOM specification for details. The situation is similar to that with nodeName. The value is a string or None.

Node.hasAttributes()
Returns true if the node has any attributes.

Node.hasChildNodes()
Returns true if the node has any child nodes.

Node.isSameNode(other)
Returns true if other refers to the same node as this node. This is especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

Note: This is based on a proposed DOM Level 3 API which is still in the "working draft" stage, but this particular interface appears uncontroversial. Changes from the W3C will not necessarily affect this method in the Python DOM interface (though any new W3C API for this would also be supported).

Node.appendChild(newChild)
Add a new child node to this node at the end of the list of children, returning newChild. If the node was already in the tree, it is removed first.

Node.insertBefore(newChild, refChild)
Insert a new child node before an existing child. It must be the case that refChild is a child of this node; if not, ValueError is raised. newChild is returned. If refChild is None, it inserts newChild at the end of the children's list.

Node.removeChild(oldChild)
Remove a child node. oldChild must be a child of this node; if not, ValueError is raised. oldChild is returned on success. If oldChild will not be used further, its unlink() method should be called.

Node.replaceChild(newChild, oldChild)
Replace an existing node with a new node. It must be the case that oldChild is a child of this node; if not, ValueError is raised.

Node.normalize()
Join adjacent text nodes so that all stretches of text are stored as single Text instances. This simplifies processing text from a DOM tree for many applications.

New in version 2.1.

Node.cloneNode(deep)
Clone this node. Setting deep means to clone all child nodes as well. This returns the clone.

NodeList Objects

A NodeList represents a sequence of nodes. These objects are used in two ways in the DOM Core recommendation: an Element object provides one as its list of child nodes, and the getElementsByTagName() and getElementsByTagNameNS() methods of Node return objects with this interface to represent query results.

The DOM Level 2 recommendation defines one method and one attribute for these objects:

NodeList.item(i)
Return the i'th item from the sequence, if there is one, or None. The index i is not allowed to be less than zero or greater than or equal to the length of the sequence.

NodeList.length
The number of nodes in the sequence.

In addition, the Python DOM interface requires that some additional support is provided to allow NodeList objects to be used as Python sequences. All NodeList implementations must include support for __len__()

and __getitem__(); this allows iteration over the NodeList in for statements and proper support for the len() built-in function.

If a DOM implementation supports modification of the document, the NodeList implementation must also support the __setitem__() and __delitem__() methods.

DocumentType Objects

Information about the notations and entities declared by a document (including the external subset if the parser uses it and can provide the information) is available from a DocumentType object. The DocumentType for a document is available from the Document object's doctype attribute; if there is no DOCTYPE declaration for the document, the document's doctype attribute will be set to None instead of an instance of this interface.

DocumentType is a specialization of Node, and adds the following attributes:

DocumentType.publicId
: The public identifier for the external subset of the document type definition. This will be a string or None.

DocumentType.systemId
: The system identifier for the external subset of the document type definition. This will be a URI as a string, or None.

DocumentType.internalSubset
: A string giving the complete internal subset from the document. This does not include the brackets which enclose the subset. If the document has no internal subset, this should be None.

DocumentType.name
: The name of the root element as given in the DOCTYPE declaration, if present.

DocumentType.entities
: This is a NamedNodeMap giving the definitions of external entities. For entity names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no entities are defined.

DocumentType.notations
: This is a NamedNodeMap giving the definitions of notations. For notation names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no notations are defined.

Document Objects

A Document represents an entire XML document, including its constituent elements, attributes, processing instructions, comments etc. Remember that it inherits properties from Node.

Document.documentElement
: The one and only root element of the document.

Document.createElement(tagName)
: Create and return a new element node. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as insertBefore() or appendChild().

Document.createElementNS(namespaceURI, tagName)
: Create and return a new element with a namespace. The tagName may have a prefix. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as insertBefore() or appendChild().

Document.createTextNode(data)
: Create and return a text node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

Document.createComment(data)
: Create and return a comment node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

Document.createProcessingInstruction(target, data)
: Create and return a processing instruction node containing the target and data passed as parameters. As with the other creation methods, this one does not insert the node into the tree.

Document.createAttribute(name)
: Create and return an attribute node. This method does not associate the attribute node with any particular element. You must use setAttributeNode() on the appropriate Element object to use the newly created attribute instance.

Document.createAttributeNS(namespaceURI, qualifiedName)
: Create and return an attribute node with a namespace. The tagName may have a prefix. This method does not associate the attribute node with any particular element. You must use setAttributeNode() on the appropriate Element object to use the newly created attribute instance.

Document.getElementsByTagName(tagName)
: Search for all descendants (direct children, children's children, etc.) with a particular element type name.

Document.getElementsByTagNameNS(namespaceURI, localName)
: Search for all descendants (direct children, children's children, etc.) with a particular namespace URI and localname. The localname is the part of the namespace after the prefix.

Element Objects

Element is a subclass of Node, so inherits all the attributes of that class.

Element.tagName
: The element type name. In a namespace-using document it may have colons in it. The value is a string.

Element.getElementsByTagName(tagName)
: Same as equivalent method in the Document class.

Element.getElementsByTagNameNS(namespaceURI, localName)
: Same as equivalent method in the Document class.

Element.hasAttribute(name)
: Returns true if the element has an attribute named by name.

Element.hasAttributeNS(namespaceURI, localName)
: Returns true if the element has an attribute named by namespaceURI and localName.

Element.getAttribute(name)
: Return the value of the attribute named by name as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

Element.getAttributeNode(attrname)
: Return the Attr node for the attribute named by attrname.

Element.getAttributeNS(namespaceURI, localName)
: Return the value of the attribute named by namespaceURI and localName as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

Element.getAttributeNodeNS(namespaceURI, localName)
: Return an attribute value as a node, given a namespaceURI and localName.

Element.removeAttribute(name)
: Remove an attribute by name. If there is no matching attribute, a NotFoundErr is raised.

Element.removeAttributeNode(oldAttr)
: Remove and return oldAttr from the attribute list, if present. If oldAttr is not present, NotFoundErr is raised.

Element.removeAttributeNS(namespaceURI, localName)
: Remove an attribute by name. Note that it uses a localName, not a qname. No exception is raised if there is no matching attribute.

Element.setAttribute(name, value)
: Set an attribute value from a string.

Element.setAttributeNode(newAttr)
: Add a new attribute node to the element, replacing an existing attribute if necessary if the name attribute matches. If a replacement occurs, the old attribute node will be returned. If newAttr is already in use, InuseAttributeErr will be raised.

Element.setAttributeNodeNS(newAttr)
: Add a new attribute node to the element, replacing an existing attribute if necessary if the namespaceURI and localName attributes match. If a replacement occurs, the old attribute node will be returned. If newAttr is already in use, InuseAttributeErr will be raised.

Element.setAttributeNS(namespaceURI, qname, value)
: Set an attribute value from a string, given a namespaceURI and a qname. Note that a qname is the whole attribute name. This is different than above.

Attr Objects

Attr inherits from Node, so inherits all its attributes.

Attr.name
: The attribute name. In a namespace-using document it may include a colon.

Attr.localName
: The part of the name following the colon if there is one, else the entire name. This is a read-only attribute.

Attr.prefix
: The part of the name preceding the colon if there is one, else the empty string.

Attr.value
: The text value of the attribute. This is a synonym for the nodeValue attribute.

NamedNodeMap Objects

NamedNodeMap does not inherit from Node.

NamedNodeMap.length
: The length of the attribute list.

NamedNodeMap.item(index)
: Return an attribute with a particular index. The order you get the attributes in is arbitrary but will be consistent for the life of a DOM. Each item is an attribute node. Get its value with the value attribute.

There are also experimental methods that give this class more mapping behavior. You can use them or you can use the standardized getAttribute*() family of methods on the Element objects.

Comment Objects

Comment represents a comment in the XML document. It is a subclass of Node, but cannot have child nodes.

Comment.data
: The content of the comment as a string. The attribute contains all characters between the leading <!-- and trailing -->, but does not include them.

Text and CDATASection Objects

The Text interface represents text in the XML document. If the parser and DOM implementation support the DOM's XML extension, portions of the text enclosed in CDATA marked sections are stored in CDATASection objects. These two interfaces are identical, but provide different values for the nodeType attribute.

These interfaces extend the Node interface. They cannot have child nodes.

Text.data
: The content of the text node as a string.

Note: The use of a CDATASection node does not indicate that the node represents a complete CDATA marked section, only that the content of the node was part of a CDATA section. A single CDATA section may be represented by more than one node in the document tree. There is no way to determine whether two adjacent CDATASection nodes represent different CDATA marked sections.

ProcessingInstruction Objects

Represents a processing instruction in the XML document; this inherits from the Node interface and cannot have child nodes.

ProcessingInstruction.target
: The content of the processing instruction up to the first whitespace character. This is a read-only attribute.

ProcessingInstruction.data
: The content of the processing instruction following the first whitespace character.

Exceptions

New in version 2.1.

The DOM Level 2 recommendation defines a single exception, DOMException, and a number of constants that allow applications to determine what sort of error occurred. DOMException instances carry a code attribute that provides the appropriate value for the specific exception.

The Python DOM interface provides the constants, but also expands the set of exceptions so that a specific exception exists for each of the exception codes defined by the DOM. The implementations must raise the appropriate specific exception, each of which carries the appropriate value for the code attribute.

exception xml.dom.DOMException
: Base exception class used for all specific DOM exceptions. This exception class cannot be directly instantiated.

exception xml.dom.DomstringSizeErr
: Raised when a specified range of text does not fit into a string. This is not known to be used in the Python DOM implementations, but may be received from DOM implementations not written in Python.

exception xml.dom.HierarchyRequestErr
: Raised when an attempt is made to insert a node where the node type is not allowed.

exception xml.dom.IndexSizeErr
: Raised when an index or size parameter to a method is negative or exceeds the allowed values.

exception xml.dom.InuseAttributeErr
: Raised when an attempt is made to insert an Attr node that is already present elsewhere in the document.

exception xml.dom.InvalidAccessErr
: Raised if a parameter or an operation is not supported on the underlying object.

exception xml.dom.InvalidCharacterErr
: This exception is raised when a string parameter contains a character that is not permitted in the context it's being used in by the XML 1.0 recommendation. For example, attempting to create an Element node with a space in the element type name will cause this error to be raised.

exception xml.dom.InvalidModificationErr
: Raised when an attempt is made to modify the type of a node.

exception xml.dom.InvalidStateErr
: Raised when an attempt is made to use an object that is not defined or is no longer usable.

exception xml.dom.NamespaceErr
: If an attempt is made to change any object in a way that is not permitted with regard to the Namespaces in XML recommendation, this exception is raised.

exception xml.dom.NotFoundErr
: Exception when a node does not exist in the referenced context. For example, NamedNodeMap. removeNamedItem() will raise this if the node passed in does not exist in the map.

exception xml.dom.NotSupportedErr
: Raised when the implementation does not support the requested type of object or operation.

exception xml.dom.NoDataAllowedErr
: This is raised if data is specified for a node which does not support data.

exception xml.dom.NoModificationAllowedErr
: Raised on attempts to modify an object where modifications are not allowed (such as for read-only nodes).

exception xml.dom.SyntaxErr
: Raised when an invalid or illegal string is specified.

exception xml.dom.WrongDocumentErr
: Raised when a node is inserted in a different document than it currently belongs to, and the implementation does not support migrating the node from one document to the other.

The exception codes defined in the DOM recommendation map to the exceptions described above according to this table:

Constant	Exception
DOMSTRING_SIZE_ERR	DomstringSizeErr
HIERARCHY_REQUEST_ERR	HierarchyRequestErr
INDEX_SIZE_ERR	IndexSizeErr
INUSE_ATTRIBUTE_ERR	InuseAttributeErr
INVALID_ACCESS_ERR	InvalidAccessErr
INVALID_CHARACTER_ERR	InvalidCharacterErr
INVALID_MODIFICATION_ERR	InvalidModificationErr
INVALID_STATE_ERR	InvalidStateErr
NAMESPACE_ERR	NamespaceErr
NOT_FOUND_ERR	NotFoundErr
NOT_SUPPORTED_ERR	NotSupportedErr
NO_DATA_ALLOWED_ERR	NoDataAllowedErr
NO_MODIFICATION_ALLOWED_ERR	NoModificationAllowedErr
SYNTAX_ERR	SyntaxErr
WRONG_DOCUMENT_ERR	WrongDocumentErr

19.8.3 Conformance

This section describes the conformance requirements and relationships between the Python DOM API, the W3C DOM recommendations, and the OMG IDL mapping for Python.

Type Mapping

The primitive IDL types used in the DOM specification are mapped to Python types according to the following table.

IDL Type	Python Type
boolean	IntegerType (with a value of 0 or 1)
int	IntegerType
long int	IntegerType
unsigned int	IntegerType

Additionally, the DOMString defined in the recommendation is mapped to a Python string or Unicode string. Applications should be able to handle Unicode whenever a string is returned from the DOM.

The IDL null value is mapped to None, which may be accepted or provided by the implementation whenever null is allowed by the API.

Accessor Methods

The mapping from OMG IDL to Python defines accessor functions for IDL attribute declarations in much the way the Java mapping does. Mapping the IDL declarations

```
readonly attribute string someValue;
        attribute string anotherValue;
```

yields three accessor functions: a "get" method for someValue (_get_someValue()), and "get" and "set" methods for anotherValue (_get_anotherValue() and _set_anotherValue()). The mapping, in particular,

does not require that the IDL attributes are accessible as normal Python attributes: object.someValue is not required to work, and may raise an AttributeError.

The Python DOM API, however, does require that normal attribute access work. This means that the typical surrogates generated by Python IDL compilers are not likely to work, and wrapper objects may be needed on the client if the DOM objects are accessed via CORBA. While this does require some additional consideration for CORBA DOM clients, the implementers with experience using DOM over CORBA from Python do not consider this a problem. Attributes that are declared readonly may not restrict write access in all DOM implementations.

In the Python DOM API, accessor functions are not required. If provided, they should take the form defined by the Python IDL mapping, but these methods are considered unnecessary since the attributes are accessible directly from Python. "Set" accessors should never be provided for readonly attributes.

The IDL definitions do not fully embody the requirements of the W3C DOM API, such as the notion of certain objects, such as the return value of getElementsByTagName(), being "live". The Python DOM API does not require implementations to enforce such requirements.

19.9 xml.dom.minidom — Minimal DOM implementation

New in version 2.0.

Source code: Lib/xml/dom/minidom.py

---

xml.dom.minidom is a minimal implementation of the Document Object Model interface, with an API similar to that in other languages. It is intended to be simpler than the full DOM and also significantly smaller. Users who are not already proficient with the DOM should consider using the xml.etree.ElementTree module for their XML processing instead.

> Warning: The xml.dom.minidom module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

DOM applications typically start by parsing some XML into a DOM. With xml.dom.minidom, this is done through the parse functions:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

The parse() function can take either a filename or an open file object.

xml.dom.minidom.parse(filename_or_file[, parser[, bufsize]])
: Return a Document from the given input. filename_or_file may be either a file name, or a file-like object. parser, if given, must be a SAX2 parser object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the parseString() function instead:

xml.dom.minidom.parseString(string[, parser])
: Return a Document that represents the string. This method creates a StringIO object for the string and passes that on to parse().

Both functions return a Document object representing the content of the document.

What the parse() and parseString() functions do is connect an XML parser with a "DOM builder" that can accept parse events from any SAX parser and convert them into a DOM tree. The name of the functions are perhaps misleading, but are easy to grasp when learning the interfaces. The parsing of the document will be completed before these functions return; it's simply that these functions do not provide a parser implementation themselves.

You can also create a Document by calling a method on a "DOM Implementation" object. You can get this object either by calling the getDOMImplementation() function in the xml.dom package or the xml.dom.minidom module. Using the implementation from the xml.dom.minidom module will always return a Document instance from the minidom implementation, while the version from xml.dom may provide an alternate implementation (this is likely if you have the PyXML package installed). Once you have a Document, you can add child nodes to it to populate the DOM:

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification. The main property of the document object is the documentElement property. It gives you the main element in the XML document: the one that holds all others. Here is an example program:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

When you are finished with a DOM tree, you may optionally call the unlink() method to encourage early cleanup of the now-unneeded objects. unlink() is an xml.dom.minidom-specific extension to the DOM API that renders the node and its descendants are essentially useless. Otherwise, Python's garbage collector will eventually take care of the objects in the tree.

See also:

Document Object Model (DOM) Level 1 Specification The W3C recommendation for the DOM supported by xml.dom.minidom.

19.9.1 DOM Objects

The definition of the DOM API for Python is given as part of the xml.dom module documentation. This section lists the differences between the API and xml.dom.minidom.

Node.unlink()
: Break internal references within the DOM so that it will be garbage collected on versions of Python without cyclic GC. Even when cyclic GC is available, using this can make large amounts of memory available sooner, so calling this on DOM objects as soon as they are no longer needed is good practice. This only needs to be called on the Document object, but may be called on child nodes to discard children of that node.

Node.writexml(writer, indent="", addindent="", newl="")
: Write XML to the writer object. The writer should have a write() method which matches that of the file object interface. The indent parameter is the indentation of the current node. The addindent parameter is the incremental indentation to use for subnodes of the current one. The newl parameter specifies the string to use to terminate newlines.

  For the Document node, an additional keyword argument encoding can be used to specify the encoding field of the XML header.

  Changed in version 2.1: The optional keyword parameters indent, addindent, and newl were added to support pretty output.

  Changed in version 2.3: For the Document node, an additional keyword argument encoding can be used to specify the encoding field of the XML header.

Node.toxml([encoding])
: Return the XML that the DOM represents as a string.

  With no argument, the XML header does not specify an encoding, and the result is Unicode string if the default encoding cannot represent all characters in the document. Encoding this string in an encoding other than UTF-8 is likely incorrect, since UTF-8 is the default encoding of XML.

  With an explicit encoding[1] argument, the result is a byte string in the specified encoding. It is recommended that this argument is always specified. To avoid UnicodeError exceptions in case of unrepresentable text data, the encoding argument should be specified as "utf-8".

  Changed in version 2.3: the encoding argument was introduced; see writexml().

Node.toprettyxml(indent="\t", newl="\n", encoding=None)
: Return a pretty-printed version of the document. indent specifies the indentation string and defaults to a tabulator; newl specifies the string emitted at the end of each line and defaults to \n.

  New in version 2.1.

  Changed in version 2.3: the encoding argument was introduced; see writexml().

The following standard DOM methods have special considerations with xml.dom.minidom:

Node.cloneNode(deep)
: Although this method was present in the version of xml.dom.minidom packaged with Python 2.0, it was seriously broken. This has been corrected for subsequent releases.

19.9.2 DOM Example

This example program is a fairly realistic example of a simple program. In this particular case, we do not take much advantage of the flexibility of the DOM.

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>
```

(continues on next page)

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

(continued from previous page)

```
<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print "<html>"
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print "</html>"

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print "<title>%s</title>" % getText(title.childNodes)

def handleSlideTitle(title):
    print "<h2>%s</h2>" % getText(title.childNodes)

def handlePoints(points):
    print "<ul>"
    for point in points:
        handlePoint(point)
    print "</ul>"

def handlePoint(point):
    print "<li>%s</li>" % getText(point.childNodes)

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print "<p>%s</p>" % getText(title.childNodes)

handleSlideshow(dom)
```

19.9.3 minidom and the DOM standard

The xml.dom.minidom module is essentially a DOM 1.0-compatible DOM with some DOM 2 features (primarily namespace features).

Usage of the DOM interface in Python is straight-forward. The following mapping rules apply:

- Interfaces are accessed through instance objects. Applications should not instantiate the classes themselves; they should use the creator functions available on the Document object. Derived interfaces support all operations (and attributes) from the base interfaces, plus any new operations.
- Operations are used as methods. Since the DOM uses only in parameters, the arguments are passed in normal order (from left to right). There are no optional arguments. void operations return None.
- IDL attributes map to instance attributes. For compatibility with the OMG IDL language mapping for Python, an attribute foo can also be accessed through accessor methods _get_foo() and _set_foo(). readonly attributes must not be changed; this is not enforced at runtime.
- The types short int, unsigned int, unsigned long long, and boolean all map to Python integer objects.
- The type DOMString maps to Python strings. xml.dom.minidom supports either byte or Unicode strings, but will normally produce Unicode strings. Values of type DOMString may also be None where allowed to have the IDL null value by the DOM specification from the W3C.
- const declarations map to variables in their respective scope (e.g. xml.dom.minidom.Node. PROCESSING_INSTRUCTION_NODE); they must not be changed.
- DOMException is currently not supported in xml.dom.minidom. Instead, xml.dom.minidom uses standard Python exceptions such as TypeError and AttributeError.
- NodeList objects are implemented using Python's built-in list type. Starting with Python 2.2, these objects provide the interface defined in the DOM specification, but with earlier versions of Python they do not support the official API. They are, however, much more "Pythonic" than the interface defined in the W3C recommendations.

The following interfaces have no implementation in xml.dom.minidom:

- DOMTimeStamp
- DocumentType (added in Python 2.1)
- DOMImplementation (added in Python 2.1)
- CharacterData
- CDATASection
- Notation
- Entity
- EntityReference
- DocumentFragment

Most of these reflect information in the XML document that is not of general utility to most DOM users.

19.10 xml.dom.pulldom — Support for building partial DOM trees

New in version 2.0.

Source code: Lib/xml/dom/pulldom.py

xml.dom.pulldom allows building only selected portions of a Document Object Model representation of a document from SAX events.

> Warning: The xml.dom.pulldom module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

class xml.dom.pulldom.PullDOM([documentFactory])
: xml.sax.handler.ContentHandler implementation that ...

class xml.dom.pulldom.DOMEventStream(stream, parser, bufsize)
: ...

class xml.dom.pulldom.SAX2DOM([documentFactory])
: xml.sax.handler.ContentHandler implementation that ...

xml.dom.pulldom.parse(stream_or_string[, parser[, bufsize]])
: ...

xml.dom.pulldom.parseString(string[, parser])
: ...

xml.dom.pulldom.default_bufsize
: Default value for the bufsize parameter to parse().

  Changed in version 2.1: The value of this variable can be changed before calling parse() and the new value will take effect.

19.10.1 DOMEventStream Objects

DOMEventStream.getEvent()
: ...

DOMEventStream.expandNode(node)
: ...

DOMEventStream.reset()
: ...

19.11 xml.sax — Support for SAX2 parsers

New in version 2.0.

The xml.sax package provides a number of modules which implement the Simple API for XML (SAX) interface for Python. The package itself provides the SAX exceptions and the convenience functions which will be most used by users of the SAX API.

> Warning: The xml.sax module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

The convenience functions are:

xml.sax.make_parser([parser_list])
: Create and return a SAX XMLReader object. The first parser found will be used. If parser_list is

provided, it must be a list of strings which name modules that have a function named create_parser(). Modules listed in parser_list will be used before modules in the default list of parsers.

xml.sax.parse(filename_or_stream, handler[, error_handler])
Create a SAX parser and use it to parse a document. The document, passed in as filename_or_stream, can be a filename or a file object. The handler parameter needs to be a SAX ContentHandler instance. If error_handler is given, it must be a SAX ErrorHandler instance; if omitted, SAXParseException will be raised on all errors. There is no return value; all work must be done by the handler passed in.

xml.sax.parseString(string, handler[, error_handler])
Similar to parse(), but parses from a buffer string received as a parameter.

A typical SAX application uses three kinds of objects: readers, handlers and input sources. "Reader" in this context is another term for parser, i.e. some piece of code that reads the bytes or characters from the input source, and produces a sequence of events. The events then get distributed to the handler objects, i.e. the reader invokes a method on the handler. A SAX application must therefore obtain a reader object, create or open the input sources, create the handlers, and connect these objects all together. As the final step of preparation, the reader is called to parse the input. During parsing, methods on the handler objects are called based on structural and syntactic events from the input data.

For these objects, only the interfaces are relevant; they are normally not instantiated by the application itself. Since Python does not have an explicit notion of interface, they are formally introduced as classes, but applications may use implementations which do not inherit from the provided classes. The InputSource, Locator, Attributes, AttributesNS, and XMLReader interfaces are defined in the module xml.sax.xmlreader. The handler interfaces are defined in xml.sax.handler. For convenience, InputSource (which is often instantiated directly) and the handler classes are also available from xml.sax. These interfaces are described below.

In addition to these classes, xml.sax provides the following exception classes.

exception xml.sax.SAXException(msg[, exception])
Encapsulate an XML error or warning. This class can contain basic error or warning information from either the XML parser or the application: it can be subclassed to provide additional functionality or to add localization. Note that although the handlers defined in the ErrorHandler interface receive instances of this exception, it is not required to actually raise the exception — it is also useful as a container for information.

When instantiated, msg should be a human-readable description of the error. The optional exception parameter, if given, should be None or an exception that was caught by the parsing code and is being passed along as information.

This is the base class for the other SAX exception classes.

exception xml.sax.SAXParseException(msg, exception, locator)
Subclass of SAXException raised on parse errors. Instances of this class are passed to the methods of the SAX ErrorHandler interface to provide information about the parse error. This class supports the SAX Locator interface as well as the SAXException interface.

exception xml.sax.SAXNotRecognizedException(msg[, exception])
Subclass of SAXException raised when a SAX XMLReader is confronted with an unrecognized feature or property. SAX applications and extensions may use this class for similar purposes.

exception xml.sax.SAXNotSupportedException(msg[, exception])
Subclass of SAXException raised when a SAX XMLReader is asked to enable a feature that is not supported, or to set a property to a value that the implementation does not support. SAX applications and extensions may use this class for similar purposes.

See also:

SAX: The Simple API for XML This site is the focal point for the definition of the SAX API. It provides a Java implementation and online documentation. Links to implementations and historical information are also available.

Module xml.sax.handler Definitions of the interfaces for application-provided objects.

Module xml.sax.saxutils Convenience functions for use in SAX applications.

Module xml.sax.xmlreader Definitions of the interfaces for parser-provided objects.

19.11.1 SAXException Objects

The SAXException exception class supports the following methods:

SAXException.getMessage()
Return a human-readable message describing the error condition.

SAXException.getException()
Return an encapsulated exception object, or None.

19.12 xml.sax.handler — Base classes for SAX handlers

New in version 2.0.

The SAX API defines four kinds of handlers: content handlers, DTD handlers, error handlers, and entity resolvers. Applications normally only need to implement those interfaces whose events they are interested in; they can implement the interfaces in a single object or in multiple objects. Handler implementations should inherit from the base classes provided in the module xml.sax.handler, so that all methods get default implementations.

class xml.sax.handler.ContentHandler
This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

class xml.sax.handler.DTDHandler
Handle DTD events.

This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

class xml.sax.handler.EntityResolver
Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.

class xml.sax.handler.ErrorHandler
Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are handled in some other way.

In addition to these classes, xml.sax.handler provides symbolic constants for the feature and property names.

xml.sax.handler.feature_namespaces

value: "http://xml.org/sax/features/namespaces"
true: Perform Namespace processing.

false: Optionally do not perform Namespace processing (implies namespace-prefixes; default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_namespace_prefixes

value: "http://xml.org/sax/features/namespace-prefixes"
true: Report the original prefixed names and attributes used for Namespace declarations.
false: Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_string_interning

value: "http://xml.org/sax/features/string-interning"
true: All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.
false: Names are not necessarily interned, although they may be (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_validation

value: "http://xml.org/sax/features/validation"
true: Report all validation errors (implies external-general-entities and external-parameter-entities).
false: Do not report validation errors.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_external_ges

value: "http://xml.org/sax/features/external-general-entities"
true: Include all external general (text) entities.
false: Do not include external general entities.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_external_pes

value: "http://xml.org/sax/features/external-parameter-entities"
true: Include all external parameter entities, including the external DTD subset.
false: Do not include any external parameter entities, even the external DTD subset.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.all_features
List of all features.

xml.sax.handler.property_lexical_handler

value: "http://xml.org/sax/properties/lexical-handler"
data type: xml.sax.sax2lib.LexicalHandler (not supported in Python 2)
description: An optional extension handler for lexical events like comments.
access: read/write

xml.sax.handler.property_declaration_handler

value: "http://xml.org/sax/properties/declaration-handler"
data type: xml.sax.sax2lib.DeclHandler (not supported in Python 2)
description: An optional extension handler for DTD-related events other than notations and unparsed entities.
access: read/write

xml.sax.handler.property_dom_node

value: "http://xml.org/sax/properties/dom-node"
data type: org.w3c.dom.Node (not supported in Python 2)
description: When parsing, the current DOM node being visited if this is a DOM iterator; when not parsing, the root DOM node for iteration.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.property_xml_string

value: "http://xml.org/sax/properties/xml-string"
data type: String
description: The literal string of characters that was the source for the current event.
access: read-only

xml.sax.handler.all_properties
List of all known property names.

19.12.1 ContentHandler Objects

Users are expected to subclass ContentHandler to support their application. The following methods are called by the parser on the appropriate events in the input document:

ContentHandler.setDocumentLocator(locator)
Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

ContentHandler.startDocument()
Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for setDocumentLocator()).

ContentHandler.endDocument()
Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

ContentHandler.startPrefixMapping(prefix, uri)
Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the feature_namespaces feature is enabled (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the startPrefixMapping() and endPrefixMapping() events supply the information to the application to expand prefixes in those contexts itself, if necessary.

Note that startPrefixMapping() and endPrefixMapping() events are not guaranteed to be properly nested relative to each-other: all startPrefixMapping() events will occur before the corresponding startElement() event, and all endPrefixMapping() events will occur after the corresponding endElement() event, but their order is not guaranteed.

ContentHandler.endPrefixMapping(prefix)
End the scope of a prefix-URI mapping.

See startPrefixMapping() for details. This event will always occur after the corresponding endElement() event, but the order of endPrefixMapping() events is not otherwise guaranteed.

ContentHandler.startElement(name, attrs)
Signals the start of an element in non-namespace mode.

The name parameter contains the raw XML 1.0 name of the element type as a string and the attrs parameter holds an object of the Attributes interface (see The Attributes Interface) containing the attributes of the element. The object passed as attrs may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the copy() method of the attrs object.

ContentHandler.endElement(name)
Signals the end of an element in non-namespace mode.

The name parameter contains the name of the element type, just as with the startElement() event.

ContentHandler.startElementNS(name, qname, attrs)
Signals the start of an element in namespace mode.

The name parameter contains the name of the element type as a (uri, localname) tuple, the qname parameter contains the raw XML 1.0 name used in the source document, and the attrs parameter holds an instance of the AttributesNS interface (see The AttributesNS Interface) containing the attributes of the element. If no namespace is associated with the element, the uri component of name will be None. The object passed as attrs may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the copy() method of the attrs object.

Parsers may set the qname parameter to None, unless the feature_namespace_prefixes feature is activated.

ContentHandler.endElementNS(name, qname)
Signals the end of an element in namespace mode.

The name parameter contains the name of the element type, just as with the startElementNS() method, likewise the qname parameter.

ContentHandler.characters(content)
Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the

characters in any single event must come from the same external entity so that the Locator provides useful information.

content may be a Unicode string or a byte string; the expat reader module produces always Unicode strings.

Note: The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use content instead of slicing content with the old offset and length parameters.

ContentHandler.ignorableWhitespace(whitespace)
Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

ContentHandler.processingInstruction(target, data)
Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

ContentHandler.skippedEntity(name)
Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the feature_external_ges and the feature_external_pes properties.

19.12.2 DTDHandler Objects

DTDHandler instances provide the following methods:

DTDHandler.notationDecl(name, publicId, systemId)
Handle a notation declaration event.

DTDHandler.unparsedEntityDecl(name, publicId, systemId, ndata)
Handle an unparsed entity declaration event.

19.12.3 EntityResolver Objects

EntityResolver.resolveEntity(publicId, systemId)
Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an InputSource to read from. The default implementation returns systemId.

19.12.4 ErrorHandler Objects

Objects with this interface are used to receive error and warning information from the XMLReader. If you create an object that implements this interface, then register the object with your XMLReader, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available: warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a SAXParseException as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

ErrorHandler.error(exception)
: Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.

ErrorHandler.fatalError(exception)
: Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.

ErrorHandler.warning(exception)
: Called when the parser presents minor warning information to the application. Parsing is expected to continue when this method returns, and document information will continue to be passed to the application. Raising an exception in this method will cause parsing to end.

19.13 xml.sax.saxutils — SAX Utilities

New in version 2.0.

The module xml.sax.saxutils contains a number of classes and functions that are commonly useful when creating SAX applications, either in direct use, or as base classes.

xml.sax.saxutils.escape(data[, entities])
: Escape '&', '<', and '>' in a string of data.

  You can escape other strings of data by passing a dictionary as the optional entities parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. The characters '&', '<' and '>' are always escaped, even if entities is provided.

xml.sax.saxutils.unescape(data[, entities])
: Unescape '&', '<', and '>' in a string of data.

  You can unescape other strings of data by passing a dictionary as the optional entities parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. '&amp', '<', and '>' are always unescaped, even if entities is provided.

  New in version 2.3.

xml.sax.saxutils.quoteattr(data[, entities])
: Similar to escape(), but also prepares data to be used as an attribute value. The return value is a quoted version of data with any additional required replacements. quoteattr() will select a quote character based on the content of data, attempting to avoid encoding any quote characters in the string. If both single- and double-quote characters are already in data, the double-quote characters will be encoded and data will be wrapped in double-quotes. The resulting string can be used directly as an attribute value:

```
>>> print "<element attr=%s>" % quoteattr("ab ' cd \" ef")
<element attr="ab ' cd &quot; ef">
```

This function is useful when generating attribute values for HTML or any SGML using the reference concrete syntax.

New in version 2.2.

class xml.sax.saxutils.XMLGenerator([out[, encoding]])
This class implements the ContentHandler interface by writing SAX events back into an XML document. In other words, using an XMLGenerator as the content handler will reproduce the original document being parsed. out should be a file-like object which will default to sys.stdout. encoding is the encoding of the output stream which defaults to 'iso-8859-1'.

class xml.sax.saxutils.XMLFilterBase(base)
This class is designed to sit between an XMLReader and the client application's event handlers. By default, it does nothing but pass requests up to the reader and events on to the handlers unmodified, but subclasses can override specific methods to modify the event stream or the configuration requests as they pass through.

xml.sax.saxutils.prepare_input_source(source[, base])
This function takes an input source and an optional base URL and returns a fully resolved InputSource object ready for reading. The input source can be given as a string, a file-like object, or an InputSource object; parsers will use this function to implement the polymorphic source argument to their parse() method.

19.14 xml.sax.xmlreader — Interface for XML parsers

New in version 2.0.

SAX parsers implement the XMLReader interface. They are implemented in a Python module, which must provide a function create_parser(). This function is invoked by xml.sax.make_parser() with no arguments to create a new parser object.

class xml.sax.xmlreader.XMLReader
Base class which can be inherited by SAX parsers.

class xml.sax.xmlreader.IncrementalParser
In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well; still parse() won't return until the entire document is processed. So these interfaces should be used if the blocking behaviour of parse() is not desirable.

When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

Note that these methods must not be called during parsing, that is, after parse has been called and before it returns.

By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0 driver writers.

class xml.sax.xmlreader.Locator
Interface for associating a SAX event with a document location. A locator object will return valid results only during calls to DocumentHandler methods; at any other time, the results are unpredictable. If information is not available, methods may return None.

class xml.sax.xmlreader.InputSource([systemId])
Encapsulation of the information needed by the XMLReader to read entities.

This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.

Applications will create objects of this class for use in the XMLReader.parse() method and for returning from EntityResolver.resolveEntity.

An InputSource belongs to the application, the XMLReader is not allowed to modify InputSource objects passed to it from the application, although it may make copies and modify those.

class xml.sax.xmlreader.AttributesImpl(attrs)
This is an implementation of the Attributes interface (see section The Attributes Interface). This is a dictionary-like object which represents the element attributes in a startElement() call. In addition to the most useful dictionary operations, it supports a number of other methods as described by the interface. Objects of this class should be instantiated by readers; attrs must be a dictionary-like object containing a mapping from attribute names to attribute values.

class xml.sax.xmlreader.AttributesNSImpl(attrs, qnames)
Namespace-aware variant of AttributesImpl, which will be passed to startElementNS(). It is derived from AttributesImpl, but understands attribute names as two-tuples of namespaceURI and localname. In addition, it provides a number of methods expecting qualified names as they appear in the original document. This class implements the AttributesNS interface (see section The AttributesNS Interface).

19.14.1 XMLReader Objects

The XMLReader interface supports the following methods:

XMLReader.parse(source)
Process an input source, producing SAX events. The source object can be a system identifier (a string identifying the input source – typically a file name or a URL), a file-like object, or an InputSource object. When parse() returns, the input is completely processed, and the parser object can be discarded or reset. As a limitation, the current implementation only accepts byte streams; processing of character streams is for further study.

XMLReader.getContentHandler()
Return the current ContentHandler.

XMLReader.setContentHandler(handler)
Set the current ContentHandler. If no ContentHandler is set, content events will be discarded.

XMLReader.getDTDHandler()
Return the current DTDHandler.

XMLReader.setDTDHandler(handler)
Set the current DTDHandler. If no DTDHandler is set, DTD events will be discarded.

XMLReader.getEntityResolver()
Return the current EntityResolver.

XMLReader.setEntityResolver(handler)
Set the current EntityResolver. If no EntityResolver is set, attempts to resolve an external entity will result in opening the system identifier for the entity, and fail if it is not available.

XMLReader.getErrorHandler()
Return the current ErrorHandler.

XMLReader.setErrorHandler(handler)
Set the current error handler. If no ErrorHandler is set, errors will be raised as exceptions, and warnings will be printed.

XMLReader.setLocale(locale)
Allow an application to set the locale for errors and warnings.

SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must raise a SAX exception. Applications may request a locale change in the middle of a parse.

XMLReader.getFeature(featurename)
Return the current setting for feature featurename. If the feature is not recognized, SAXNotRecognizedException is raised. The well-known featurenames are listed in the module xml.sax.handler.

XMLReader.setFeature(featurename, value)
Set the featurename to value. If the feature is not recognized, SAXNotRecognizedException is raised. If the feature or its setting is not supported by the parser, SAXNotSupportedException is raised.

XMLReader.getProperty(propertyname)
Return the current setting for property propertyname. If the property is not recognized, a SAXNotRecognizedException is raised. The well-known propertynames are listed in the module xml.sax.handler.

XMLReader.setProperty(propertyname, value)
Set the propertyname to value. If the property is not recognized, SAXNotRecognizedException is raised. If the property or its setting is not supported by the parser, SAXNotSupportedException is raised.

19.14.2 IncrementalParser Objects

Instances of IncrementalParser offer the following additional methods:

IncrementalParser.feed(data)
Process a chunk of data.

IncrementalParser.close()
Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.

IncrementalParser.reset()
This method is called after close has been called to reset the parser so that it is ready to parse new documents. The results of calling parse or feed after close without calling reset are undefined.

19.14.3 Locator Objects

Instances of Locator provide these methods:

Locator.getColumnNumber()
Return the column number where the current event begins.

Locator.getLineNumber()
Return the line number where the current event begins.

Locator.getPublicId()
Return the public identifier for the current event.

Locator.getSystemId()
Return the system identifier for the current event.

19.14.4 InputSource Objects

InputSource.setPublicId(id)
Sets the public identifier of this InputSource.

InputSource.getPublicId()
Returns the public identifier of this InputSource.

InputSource.setSystemId(id)
Sets the system identifier of this InputSource.

InputSource.getSystemId()
Returns the system identifier of this InputSource.

InputSource.setEncoding(encoding)
Sets the character encoding of this InputSource.

The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).

The encoding attribute of the InputSource is ignored if the InputSource also contains a character stream.

InputSource.getEncoding()
Get the character encoding of this InputSource.

InputSource.setByteStream(bytefile)
Set the byte stream (a Python file-like object which does not perform byte-to-character conversion) for this input source.

The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.

If the application knows the character encoding of the byte stream, it should set it with the setEncoding method.

InputSource.getByteStream()
Get the byte stream for this input source.

The getEncoding method will return the character encoding for this byte stream, or None if unknown.

InputSource.setCharacterStream(charfile)
Set the character stream for this input source. (The stream must be a Python 1.6 Unicode-wrapped file-like that performs conversion to Unicode strings.)

If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

InputSource.getCharacterStream()
Get the character stream for this input source.

19.14.5 The Attributes Interface

Attributes objects implement a portion of the mapping protocol, including the methods copy(), get(), has_key(), items(), keys(), and values(). The following methods are also provided:

Attributes.getLength()
Return the number of attributes.

Attributes.getNames()
Return the names of the attributes.

Attributes.getType(name)
: Returns the type of the attribute name, which is normally `'CDATA'`.

Attributes.getValue(name)
: Return the value of attribute name.

19.14.6 The AttributesNS Interface

This interface is a subtype of the Attributes interface (see section The Attributes Interface). All methods supported by that interface are also available on AttributesNS objects.

The following methods are also available:

AttributesNS.getValueByQName(name)
: Return the value for a qualified name.

AttributesNS.getNameByQName(name)
: Return the (namespace, localname) pair for a qualified name.

AttributesNS.getQNameByName(name)
: Return the qualified name for a (namespace, localname) pair.

AttributesNS.getQNames()
: Return the qualified names of all attributes.

19.15 xml.parsers.expat — Fast XML parsing using Expat

> Warning: The pyexpat module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

New in version 2.0.

The xml.parsers.expat module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, xmlparser, that represents the current state of an XML parser. After an xmlparser object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the pyexpat module to provide access to the Expat parser. Direct use of the pyexpat module is deprecated.

This module provides one exception and one type object:

exception xml.parsers.expat.ExpatError
: The exception raised when Expat reports an error. See section ExpatError Exceptions for more information on interpreting Expat errors.

exception xml.parsers.expat.error
: Alias for ExpatError.

xml.parsers.expat.XMLParserType
: The type of the return values from the ParserCreate() function.

The xml.parsers.expat module contains two functions:

xml.parsers.expat.ErrorString(errno)
: Returns an explanatory string for a given error number errno.

xml.parsers.expat.ParserCreate([encoding[, namespace_separator]])

Creates and returns a new xmlparser object. encoding, if specified, must be a string naming the encoding used by the XML data. Expat doesn't support as many encodings as Python does, and its repertoire of encodings can't be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If encoding[1] is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for namespace_separator. The value must be a one-character string; a ValueError will be raised if the string has an illegal length (None is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers StartElementHandler and EndElementHandler will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (chr(0)) then the namespace URI and the local part will be concatenated without any separator.

For example, if namespace_separator is set to a space character (`' '`) and the following document is parsed:

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

StartElementHandler will receive the following strings for each element:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

Due to limitations in the Expat library used by pyexpat, the xmlparser instance returned can only be used to parse a single XML document. Call ParserCreate for each document to provide unique parser instances.

See also:

The Expat XML Parser Home page of the Expat project.

19.15.1 XMLParser Objects

xmlparser objects have the following methods:

xmlparser.Parse(data[, isfinal])

Parses the contents of the string data, calling the appropriate handler functions to process the parsed data. isfinal must be true on the final call to this method; it allows the parsing of a single file in fragments, not the submission of multiple files. data can be the empty string at any time.

xmlparser.ParseFile(file)

Parse XML data reading from the object file. file only needs to provide the read(nbytes) method, returning the empty string when there's no more data.

xmlparser.SetBase(base)

Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

relative identifiers is left to the application: this value will be passed through as the base argument to the ExternalEntityRefHandler(), NotationDeclHandler(), and UnparsedEntityDeclHandler() functions.

xmlparser.GetBase()
Returns a string containing the base set by a previous call to SetBase(), or None if SetBase() hasn't been called.

xmlparser.GetInputContext()
Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is None.

New in version 2.1.

xmlparser.ExternalEntityParserCreate(context[, encoding])
Create a "child" parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The context parameter should be the string passed to the ExternalEntityRefHandler() handler function, described below. The child parser is created with the ordered_attributes, returns_unicode and specified_attributes set to the values of this parser.

xmlparser.SetParamEntityParsing(flag)
Control parsing of parameter entities (including the external DTD subset). Possible flag values are XML_PARAM_ENTITY_PARSING_NEVER, XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE and XML_PARAM_ENTITY_PARSING_ALWAYS. Return true if setting the flag was successful.

xmlparser.UseForeignDTD([flag])
Calling this with a true value for flag (the default) will cause Expat to call the ExternalEntityRefHandler with None for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the ExternalEntityRefHandler will still be called, but the StartDoctypeDeclHandler and EndDoctypeDeclHandler will not be called.

Passing a false value for flag will cancel a previous call that passed a true value, but otherwise has no effect.

This method can only be called before the Parse() or ParseFile() methods are called; calling it after either of those have been called causes ExpatError to be raised with the code attribute set to errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING.

New in version 2.3.

xmlparser objects have the following attributes:

xmlparser.buffer_size
The size of the buffer used when buffer_text is true. A new buffer size can be set by assigning a new integer value to this attribute. When the size is changed, the buffer will be flushed.

New in version 2.3.

Changed in version 2.6: The buffer size can now be changed.

xmlparser.buffer_text
Setting this to true causes the xmlparser object to buffer textual content returned by Expat to avoid multiple calls to the CharacterDataHandler() callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time.

New in version 2.3.

xmlparser.buffer_used
If buffer_text is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when buffer_text is false.

New in version 2.3.

xmlparser.ordered_attributes
: Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented: the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false; it may be changed at any time.

  New in version 2.1.

xmlparser.returns_unicode
: If this attribute is set to a non-zero integer, the handler functions will be passed Unicode strings. If returns_unicode is False, 8-bit strings containing UTF-8 encoded data will be passed to the handlers. This is True by default when Python is built with Unicode support.

  Changed in version 1.6: Can be changed at any time to affect the result type.

xmlparser.specified_attributes
: If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to comply with the standards for the behavior of XML processors. By default, this attribute is false; it may be changed at any time.

  New in version 2.1.

The following attributes contain values relating to the most recent error encountered by an xmlparser object, and will only have correct values once a call to Parse() or ParseFile() has raised an xml.parsers.expat.ExpatError exception.

xmlparser.ErrorByteIndex
: Byte index at which an error occurred.

xmlparser.ErrorCode
: Numeric code specifying the problem. This value can be passed to the ErrorString() function, or compared to one of the constants defined in the errors object.

xmlparser.ErrorColumnNumber
: Column number at which an error occurred.

xmlparser.ErrorLineNumber
: Line number at which an error occurred.

The following attributes contain values relating to the current parse location in an xmlparser object. During a callback reporting a parse event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback).

New in version 2.4.

xmlparser.CurrentByteIndex
: Current byte index in the parser input.

xmlparser.CurrentColumnNumber
: Current column number in the parser input.

xmlparser.CurrentLineNumber
: Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an xmlparser object o, use o.handlername = func. handlername must be taken from the following list, and func must be a callable object accepting the correct number of arguments. The arguments are all strings, unless otherwise stated.

xmlparser.XmlDeclHandler(version, encoding, standalone)
: Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional "standalone" declaration. version and encoding will be strings of the type dictated by the returns_unicode attribute, and standalone will be 1 if the document is declared standalone, 0 if it is declared not to be standalone, or -1 if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer.

  New in version 2.1.

xmlparser.StartDoctypeDeclHandler(doctypeName, systemId, publicId, has_internal_subset)
: Called when Expat begins parsing the document type declaration (<!DOCTYPE ...). The doctypeName is provided exactly as presented. The systemId and publicId parameters give the system and public identifiers if specified, or None if omitted. has_internal_subset will be true if the document contains and internal document declaration subset. This requires Expat version 1.2 or newer.

xmlparser.EndDoctypeDeclHandler()
: Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

xmlparser.ElementDeclHandler(name, model)
: Called once for each element type declaration. name is the name of the element type, and model is a representation of the content model.

xmlparser.AttlistDeclHandler(elname, attname, type, default, required)
: Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. elname is the name of the element to which the declaration applies and attname is the name of the attribute declared. The attribute type is a string passed as type; the possible values are 'CDATA', 'ID', 'IDREF', ... default gives the default value for the attribute used when the attribute is not specified by the document instance, or None if there is no default value (#IMPLIED values). If the attribute is required to be given in the document instance, required will be true. This requires Expat version 1.95.0 or newer.

xmlparser.StartElementHandler(name, attributes)
: Called for the start of every element. name is a string containing the element name, and attributes is a dictionary mapping attribute names to their values.

xmlparser.EndElementHandler(name)
: Called for the end of every element.

xmlparser.ProcessingInstructionHandler(target, data)
: Called for every processing instruction.

xmlparser.CharacterDataHandler(data)
: Called for character data. This will be called for normal character data, CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the StartCdataSectionHandler, EndCdataSectionHandler, and ElementDeclHandler callbacks to collect the required information.

xmlparser.UnparsedEntityDeclHandler(entityName, base, systemId, publicId, notationName)
: Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent versions, use EntityDeclHandler instead. (The underlying function in the Expat library has been declared obsolete.)

xmlparser.EntityDeclHandler(entityName, is_parameter_entity, value, base, systemId, publicId, notationName)
: Called for all entity declarations. For parameter and internal entities, value will be a string giving the declared contents of the entity; this will be None for external entities. The notationName parameter will be None for parsed entities, and the name of the notation for unparsed entities. is_parameter_entity

will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library.

New in version 2.1.

xmlparser.NotationDeclHandler(notationName, base, systemId, publicId)
Called for notation declarations. notationName, base, and systemId, and publicId are strings if given. If the public identifier is omitted, publicId will be None.

xmlparser.StartNamespaceDeclHandler(prefix, uri)
Called when an element contains a namespace declaration. Namespace declarations are processed before the StartElementHandler is called for the element on which declarations are placed.

xmlparser.EndNamespaceDeclHandler(prefix)
Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the StartNamespaceDeclHandler was called to indicate the start of each namespace declaration's scope. Calls to this handler are made after the corresponding EndElementHandler for the end of the element.

xmlparser.CommentHandler(data)
Called for comments. data is the text of the comment, excluding the leading '<!--' and trailing '-->'.

xmlparser.StartCdataSectionHandler()
Called at the start of a CDATA section. This and EndCdataSectionHandler are needed to be able to identify the syntactical start and end for CDATA sections.

xmlparser.EndCdataSectionHandler()
Called at the end of a CDATA section.

xmlparser.DefaultHandler(data)
Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

xmlparser.DefaultHandlerExpand(data)
This is the same as the DefaultHandler(), but doesn't inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

xmlparser.NotStandaloneHandler()
Called if the XML document hasn't been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to yes in an XML declaration. If this handler returns 0, then the parser will raise an XML_ERROR_NOT_STANDALONE error. If this handler is not set, no exception is raised by the parser for this condition.

xmlparser.ExternalEntityRefHandler(context, base, systemId, publicId)
Called for references to external entities. base is the current base, as set by a previous call to SetBase(). The public and system identifiers, systemId and publicId, are strings if given; if the public identifier is not given, publicId will be None. The context value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using ExternalEntityParserCreate(context), initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns 0, the parser will raise an XML_ERROR_EXTERNAL_ENTITY_HANDLING error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the DefaultHandler callback, if provided.

19.15.2 ExpatError Exceptions

ExpatError exceptions have a number of interesting attributes:

ExpatError.code
: Expat's internal error number for the specific error. This will match one of the constants defined in the errors object from this module.

  New in version 2.1.

ExpatError.lineno
: Line number on which the error was detected. The first line is numbered 1.

  New in version 2.1.

ExpatError.offset
: Character offset into the line where the error occurred. The first column is numbered 0.

  New in version 2.1.

19.15.3 Example

The following program defines three handlers that just print out their arguments.

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print 'Start element:', name, attrs
def end_element(name):
    print 'End element:', name
def char_data(data):
    print 'Character data:', repr(data)

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

The output from this program is:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

19.15.4 Content Model Descriptions

Content models are described using nested tuples. Each tuple contains four values: the type, the quantifier, the name, and a tuple of children. Children are simply additional content model descriptions.

The values of the first two fields are constants defined in the model object of the xml.parsers.expat module. These constants can be collected in two groups: the model type group and the quantifier group.

The constants in the model type group are:

xml.parsers.expat.XML_CTYPE_ANY
: The element named by the model name was declared to have a content model of ANY.

xml.parsers.expat.XML_CTYPE_CHOICE
: The named element allows a choice from a number of options; this is used for content models such as (A | B | C).

xml.parsers.expat.XML_CTYPE_EMPTY
: Elements which are declared to be EMPTY have this model type.

xml.parsers.expat.XML_CTYPE_MIXED

xml.parsers.expat.XML_CTYPE_NAME

xml.parsers.expat.XML_CTYPE_SEQ
: Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as (A, B, C).

The constants in the quantifier group are:

xml.parsers.expat.XML_CQUANT_NONE
: No modifier is given, so it can appear exactly once, as for A.

xml.parsers.expat.XML_CQUANT_OPT
: The model is optional: it can appear once or not at all, as for A?.

xml.parsers.expat.XML_CQUANT_PLUS
: The model must occur one or more times (like A+).

xml.parsers.expat.XML_CQUANT_REP
: The model must occur zero or more times, as for A*.

19.15.5 Expat error constants

The following constants are provided in the errors object of the xml.parsers.expat module. These constants are useful in interpreting some of the attributes of the ExpatError exception objects raised when an error has occurred.

The errors object has the following attributes:

xml.parsers.expat.XML_ERROR_ASYNC_ENTITY

xml.parsers.expat.XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF
: An entity reference in an attribute value referred to an external entity instead of an internal entity.

xml.parsers.expat.XML_ERROR_BAD_CHAR_REF
: A character reference referred to a character which is illegal in XML (for example, character 0, or '&#0;').

xml.parsers.expat.XML_ERROR_BINARY_ENTITY_REF
: An entity reference referred to an entity which was declared with a notation, so cannot be parsed.

xml.parsers.expat.XML_ERROR_DUPLICATE_ATTRIBUTE
An attribute was used more than once in a start tag.

xml.parsers.expat.XML_ERROR_INCORRECT_ENCODING

xml.parsers.expat.XML_ERROR_INVALID_TOKEN
Raised when an input byte could not properly be assigned to a character; for example, a NUL byte (value 0) in a UTF-8 input stream.

xml.parsers.expat.XML_ERROR_JUNK_AFTER_DOC_ELEMENT
Something other than whitespace occurred after the document element.

xml.parsers.expat.XML_ERROR_MISPLACED_XML_PI
An XML declaration was found somewhere other than the start of the input data.

xml.parsers.expat.XML_ERROR_NO_ELEMENTS
The document contains no elements (XML requires all documents to contain exactly one top-level element)..

xml.parsers.expat.XML_ERROR_NO_MEMORY
Expat was not able to allocate memory internally.

xml.parsers.expat.XML_ERROR_PARAM_ENTITY_REF
A parameter entity reference was found where it was not allowed.

xml.parsers.expat.XML_ERROR_PARTIAL_CHAR
An incomplete character was found in the input.

xml.parsers.expat.XML_ERROR_RECURSIVE_ENTITY_REF
An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.

xml.parsers.expat.XML_ERROR_SYNTAX
Some unspecified syntax error was encountered.

xml.parsers.expat.XML_ERROR_TAG_MISMATCH
An end tag did not match the innermost open start tag.

xml.parsers.expat.XML_ERROR_UNCLOSED_TOKEN
Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.

xml.parsers.expat.XML_ERROR_UNDEFINED_ENTITY
A reference was made to an entity which was not defined.

xml.parsers.expat.XML_ERROR_UNKNOWN_ENCODING
The document encoding is not supported by Expat.

xml.parsers.expat.XML_ERROR_UNCLOSED_CDATA_SECTION
A CDATA marked section was not closed.

xml.parsers.expat.XML_ERROR_EXTERNAL_ENTITY_HANDLING

xml.parsers.expat.XML_ERROR_NOT_STANDALONE
The parser determined that the document was not “standalone” though it declared itself to be in the XML declaration, and the NotStandaloneHandler was set and returned 0.

xml.parsers.expat.XML_ERROR_UNEXPECTED_STATE

xml.parsers.expat.XML_ERROR_ENTITY_DECLARED_IN_PE

xml.parsers.expat.XML_ERROR_FEATURE_REQUIRES_XML_DTD
An operation was requested that requires DTD support to be compiled in, but Expat was configured

without DTD support. This should never be reported by a standard build of the xml.parsers.expat module.

xml.parsers.expat.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING
: A behavioral change was requested after parsing started that can only be changed before parsing has started. This is (currently) only raised by UseForeignDTD().

xml.parsers.expat.XML_ERROR_UNBOUND_PREFIX
: An undeclared prefix was found when namespace processing was enabled.

xml.parsers.expat.XML_ERROR_UNDECLARING_PREFIX
: The document attempted to remove the namespace declaration associated with a prefix.

xml.parsers.expat.XML_ERROR_INCOMPLETE_PE
: A parameter entity contained incomplete markup.

xml.parsers.expat.XML_ERROR_XML_DECL
: The document contained no document element at all.

xml.parsers.expat.XML_ERROR_TEXT_DECL
: There was an error parsing a text declaration in an external entity.

xml.parsers.expat.XML_ERROR_PUBLICID
: Characters were found in the public id that are not allowed.

xml.parsers.expat.XML_ERROR_SUSPENDED
: The requested operation was made on a suspended parser, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.XML_ERROR_NOT_SUSPENDED
: An attempt to resume the parser was made when the parser had not been suspended.

xml.parsers.expat.XML_ERROR_ABORTED
: This should not be reported to Python applications.

xml.parsers.expat.XML_ERROR_FINISHED
: The requested operation was made on a parser which was finished parsing input, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.XML_ERROR_SUSPEND_PE

CHAPTER

TWENTY

INTERNET PROTOCOLS AND SUPPORT

The modules described in this chapter implement Internet protocols and support for related technology. They are all implemented in Python. Most of these modules require the presence of the system-dependent module socket, which is currently supported on most popular platforms. Here is an overview:

20.1 webbrowser — Convenient Web-browser controller

Source code: Lib/webbrowser.py

The webbrowser module provides a high-level interface to allow displaying Web-based documents to users. Under most circumstances, simply calling the open() function from this module will do the right thing.

Under Unix, graphical browsers are preferred under X11, but text-mode browsers will be used if graphical browsers are not available or an X11 display isn't available. If text-mode browsers are used, the calling process will block until the user exits the browser.

If the environment variable BROWSER exists, it is interpreted to override the platform default list of browsers, as an os.pathsep-separated list of browsers to try in order. When the value of a list part contains the string %s, then it is interpreted as a literal browser command line to be used with the argument URL substituted for %s; if the part does not contain %s, it is simply interpreted as the name of the browser to launch.[1]

For non-Unix platforms, or when a remote browser is available on Unix, the controlling process will not wait for the user to finish with the browser, but allow the remote browser to maintain its own windows on the display. If remote browsers are not available on Unix, the controlling process will launch a new browser and wait.

The script webbrowser can be used as a command-line interface for the module. It accepts a URL as the argument. It accepts the following optional parameters: -n opens the URL in a new browser window, if possible; -t opens the URL in a new browser page ("tab"). The options are, naturally, mutually exclusive. Usage example:

```
python -m webbrowser -t "http://www.python.org"
```

The following exception is defined:

exception webbrowser.Error
: Exception raised when a browser control error occurs.

The following functions are defined:

[1] Executables named here without a full path will be searched in the directories given in the PATH environment variable.

webbrowser.open(url, new=0, autoraise=True)
: Display url using the default browser. If new is 0, the url is opened in the same browser window if possible. If new is 1, a new browser window is opened if possible. If new is 2, a new browser page ("tab") is opened if possible. If autoraise is True, the window is raised if possible (note that under many window managers this will occur regardless of the setting of this variable).

  Note that on some platforms, trying to open a filename using this function, may work and start the operating system's associated program. However, this is neither supported nor portable.

  Changed in version 2.5: new can now be 2.

webbrowser.open_new(url)
: Open url in a new window of the default browser, if possible, otherwise, open url in the only browser window.

webbrowser.open_new_tab(url)
: Open url in a new page ("tab") of the default browser, if possible, otherwise equivalent to open_new().

  New in version 2.5.

webbrowser.get([name])
: Return a controller object for the browser type name. If name is empty, return a controller for a default browser appropriate to the caller's environment.

webbrowser.register(name, constructor[, instance])
: Register the browser type name. Once a browser type is registered, the get() function can return a controller for that browser type. If instance is not provided, or is None, constructor will be called without parameters to create an instance when needed. If instance is provided, constructor will never be called, and may be None.

  This entry point is only useful if you plan to either set the BROWSER variable or call get() with a nonempty argument matching the name of a handler you declare.

A number of browser types are predefined. This table gives the type names that may be passed to the get() function and the corresponding instantiations for the controller classes, all defined in this module.

Type Name	Class Name	Notes
'mozilla'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'netscape'	Mozilla('netscape')	
'galeon'	Galeon('galeon')	
'epiphany'	Galeon('epiphany')	
'skipstone'	BackgroundBrowser('skipstone')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'mosaic'	BackgroundBrowser('mosaic')	
'opera'	Opera()	
'grail'	Grail()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'macosx'	MacOSX('default')	(3)
'safari'	MacOSX('safari')	(3)
'google-chrome'	Chrome('google-chrome')	(4)
'chrome'	Chrome('chrome')	(4)
'chromium'	Chromium('chromium')	(4)
'chromium-browser'	Chromium('chromium-browser')	(4)

Notes:

(1) "Konqueror" is the file manager for the KDE desktop environment for Unix, and only makes sense to use if KDE is running. Some way of reliably detecting KDE would be nice; the KDEDIR variable is not sufficient. Note also that the name "kfm" is used even when using the konqueror command with KDE 2 — the implementation selects the best strategy for running Konqueror.

(2) Only on Windows platforms.

(3) Only on Mac OS X platform.

(4) Support for Chrome/Chromium has been added in version 2.7.5.

Here are some simple examples:

```
url = 'http://www.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url + 'doc/')

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

20.1.1 Browser Controller Objects

Browser controllers provide these methods which parallel three of the module-level convenience functions:

controller.open(url, new=0, autoraise=True)
Display url using the browser handled by this controller. If new is 1, a new browser window is opened if possible. If new is 2, a new browser page ("tab") is opened if possible.

controller.open_new(url)
Open url in a new window of the browser handled by this controller, if possible, otherwise, open url in the only browser window. Alias open_new().

controller.open_new_tab(url)
Open url in a new page ("tab") of the browser handled by this controller, if possible, otherwise equivalent to open_new().

New in version 2.5.

20.2 cgi — Common Gateway Interface support

Source code: Lib/cgi.py

Support module for Common Gateway Interface (CGI) scripts.

This module defines a number of utilities for use by CGI scripts written in Python.

20.2.1 Introduction

A CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML <FORM> or <ISINDEX> element.

Most often, CGI scripts live in the server's special cgi-bin directory. The HTTP server places all sorts of information about the request (such as the client's hostname, the requested URL, the query string, and lots of other goodies) in the script's shell environment, executes the script, and sends the script's output back to the client.

The script's input is connected to the client too, and sometimes the form data is read this way; at other times the form data is passed via the "query string" part of the URL. This module is intended to take care of the different cases and provide a simpler interface to the Python script. It also provides a number of utilities that help in debugging scripts, and the latest addition is support for file uploads from a form (if your browser supports it).

The output of a CGI script should consist of two sections, separated by a blank line. The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this:

```
print "Content-Type: text/html"     # HTML is following
print                               # blank line, end of headers
```

The second section is usually HTML, which allows the client software to display nicely formatted text with header, in-line images, etc. Here's Python code that prints a simple piece of HTML:

```
print "<TITLE>CGI script output</TITLE>"
print "<H1>This is my first CGI script</H1>"
print "Hello, world!"
```

20.2.2 Using the cgi module

Begin by writing import cgi. Do not use from cgi import * — the module defines all sorts of names for its own use or for backward compatibility that you don't want in your namespace.

When you write a new script, consider adding these lines:

```
import cgitb
cgitb.enable()
```

This activates a special exception handler that will display detailed reports in the Web browser if any errors occur. If you'd rather not show the guts of your program to users of your script, you can have the reports saved to files instead, with code like this:

```
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

It's very helpful to use this feature during script development. The reports produced by cgitb provide information that can save you a lot of time in tracking down bugs. You can always remove the cgitb line later when you have tested your script and are confident that it works correctly.

To get at submitted form data, it's best to use the FieldStorage class. The other classes defined in this module are provided mostly for backward compatibility. Instantiate it exactly once, without arguments. This reads the form contents from standard input or the environment (depending on the value of various environment variables set according to the CGI standard). Since it may consume standard input, it should be instantiated only once.

The FieldStorage instance can be indexed like a Python dictionary. It allows membership testing with the in operator, and also supports the standard dictionary method keys() and the built-in function len(). Form fields containing empty strings are ignored and do not appear in the dictionary; to keep such values, provide a true value for the optional keep_blank_values keyword parameter when creating the FieldStorage instance.

For instance, the following code (which assumes that the Content-Type header and blank line have already been printed) checks that the fields name and addr are both set to a non-empty string:

```
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print "<H1>Error</H1>"
    print "Please fill in the name and addr fields."
    return
print "<p>name:", form["name"].value
print "<p>addr:", form["addr"].value
...further form processing here...
```

Here the fields, accessed through form[key], are themselves instances of FieldStorage (or MiniFieldStorage, depending on the form encoding). The value attribute of the instance yields the string value of the field. The getvalue() method returns this string value directly; it also accepts an optional second argument as a default to return if the requested key is not present.

If the submitted form data contains more than one field with the same name, the object retrieved by form[key] is not a FieldStorage or MiniFieldStorage instance but a list of such instances. Similarly, in this situation, form.getvalue(key) would return a list of strings. If you expect this possibility (when your HTML form contains multiple fields with the same name), use the getlist() method, which always returns a list of values (so that you do not need to special-case the single item case). For example, this code concatenates any number of username fields, separated by commas:

```
value = form.getlist("username")
usernames = ",".join(value)
```

If a field represents an uploaded file, accessing the value via the value attribute or the getvalue() method reads the entire file in memory as a string. This may not be what you want. You can test for an uploaded file by testing either the filename attribute or the file attribute. You can then read the data at leisure from the file attribute:

```
fileitem = form["userfile"]
if fileitem.file:
    # It 's an uploaded file; count lines
    linecount = 0
    while 1:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

If an error is encountered when obtaining the contents of an uploaded file (for example, when the user interrupts the form submission by clicking on a Back or Cancel button) the done attribute of the object for the field will be set to the value -1.

The file upload draft standard entertains the possibility of uploading multiple files from one field (using a recursive multipart/* encoding). When this occurs, the item will be a dictionary-like FieldStorage item. This can be determined by testing its type attribute, which should be multipart/form-data (or perhaps another MIME type matching multipart/*). In this case, it can be iterated over recursively just like the top-level form object.

When a form is submitted in the "old" format (as the query string or as a single data part of type application/x-www-form-urlencoded), the items will actually be instances of the class MiniFieldStorage. In this case, the list, file, and filename attributes are always None.

A form submitted via POST that also has a query string will contain both FieldStorage and MiniFieldStorage items.

20.2.3 Higher Level Interface

New in version 2.2.

The previous section explains how to read CGI form data using the FieldStorage class. This section describes a higher level interface which was added to this class to allow one to do it in a more readable and intuitive way. The interface doesn't make the techniques described in previous sections obsolete — they are still useful to process file uploads efficiently, for example.

The interface consists of two simple methods. Using the methods you can process form data in a generic way, without the need to worry whether only one or more values were posted under one name.

In the previous section, you learned to write following code anytime you expected a user to post more than one value under one name:

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

This situation is common for example when a form contains a group of multiple checkboxes with the same name:

```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

In most situations, however, there's only one form control with a particular name in a form and then you expect and need only one value associated with this name. So you write a script containing for example this code:

```
user = form.getvalue("user").upper()
```

The problem with the code is that you should never expect that a client will provide valid input to your scripts. For example, if a curious user appends another user=foo pair to the query string, then the script would crash, because in this situation the getvalue("user") method call returns a list instead of a string. Calling the upper() method on a list is not valid (since lists do not have a method of this name) and results in an AttributeError exception.

Therefore, the appropriate way to read form data values was to always use the code which checks whether the obtained value is a single value or a list of values. That's annoying and leads to less readable scripts.

A more convenient approach is to use the methods getfirst() and getlist() provided by this higher level interface.

FieldStorage.getfirst(name[, default])
: This method always returns only one value associated with form field name. The method returns only the first value in case that more values were posted under such name. Please note that the order in which the values are received may vary from browser to browser and should not be counted on.[1] If no such form field or value exists then the method returns the value specified by the optional parameter default. This parameter defaults to None if not specified.

FieldStorage.getlist(name)
: This method always returns a list of values associated with form field name. The method returns an empty list if no such form field or value exists for name. It returns a list consisting of one item if only one such value exists.

Using these methods you can write nice compact code:

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

20.2.4 Old classes

Deprecated since version 2.6: These classes, present in earlier versions of the cgi module, are still supported for backward compatibility. New applications should use the FieldStorage class.

SvFormContentDict stores single value form content as dictionary; it assumes each field name occurs in the form only once.

FormContentDict stores multiple value form content as a dictionary (the form items are lists of values). Useful if your form contains multiple fields with the same name.

Other classes (FormContent, InterpFormContentDict) are present for backwards compatibility with really old applications only.

20.2.5 Functions

These are useful if you want more control, or if you want to employ some of the algorithms implemented in this module in other circumstances.

[1] Note that some recent versions of the HTML specification do state what order the field values should be supplied in, but knowing whether a request was received from a conforming browser, or even from a browser at all, is tedious and error-prone.

cgi.parse(fp[, environ[, keep_blank_values[, strict_parsing]]])
Parse a query in the environment or from a file (the file defaults to sys.stdin and environment defaults to os.environ). The keep_blank_values and strict_parsing parameters are passed to urlparse.parse_qs() unchanged.

cgi.parse_qs(qs[, keep_blank_values[, strict_parsing[, max_num_fields]]])
This function is deprecated in this module. Use urlparse.parse_qs() instead. It is maintained here only for backward compatibility.

cgi.parse_qsl(qs[, keep_blank_values[, strict_parsing[, max_num_fields]]])
This function is deprecated in this module. Use urlparse.parse_qsl() instead. It is maintained here only for backward compatibility.

cgi.parse_multipart(fp, pdict)
Parse input of type multipart/form-data (for file uploads). Arguments are fp for the input file and pdict for a dictionary containing other parameters in the Content-Type header.

Returns a dictionary just like urlparse.parse_qs() keys are the field names, each value is a list of values for that field. This is easy to use but not much good if you are expecting megabytes to be uploaded — in that case, use the FieldStorage class instead which is much more flexible.

Note that this does not parse nested multipart parts — use FieldStorage for that.

cgi.parse_header(string)
Parse a MIME header (such as Content-Type) into a main value and a dictionary of parameters.

cgi.test()
Robust test CGI script, usable as main program. Writes minimal HTTP headers and formats all information provided to the script in HTML form.

cgi.print_environ()
Format the shell environment in HTML.

cgi.print_form(form)
Format a form in HTML.

cgi.print_directory()
Format the current directory in HTML.

cgi.print_environ_usage()
Print a list of useful (used by CGI) environment variables in HTML.

cgi.escape(s[, quote])
Convert the characters '&', '<' and '>' in string s to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag quote is true, the quotation mark character (") is also translated; this helps for inclusion in an HTML attribute value delimited by double quotes, as in <a href="...">. Note that single quotes are never translated.

If the value to be quoted might include single- or double-quote characters, or both, consider using the quoteattr() function in the xml.sax.saxutils module instead.

20.2.6 Caring about security

There's one important rule: if you invoke an external program (via the os.system() or os.popen() functions. or others with similar functionality), make very sure you don't pass arbitrary strings received from the client to the shell. This is a well-known security hole whereby clever hackers anywhere on the Web can exploit a gullible CGI script to invoke arbitrary shell commands. Even parts of the URL or field names cannot be trusted, since the request doesn't have to come from your form!

To be on the safe side, if you must pass a string gotten from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores, and periods.

20.2.7 Installing your CGI script on a Unix system

Read the documentation for your HTTP server and check with your local system administrator to find the directory where CGI scripts should be installed; usually this is in a directory cgi-bin in the server tree.

Make sure that your script is readable and executable by "others"; the Unix file mode should be 0755 octal (use chmod 0755 filename). Make sure that the first line of the script contains #! starting in column 1 followed by the pathname of the Python interpreter, for instance:

```
#!/usr/local/bin/python
```

Make sure the Python interpreter exists and is executable by "others".

Make sure that any files your script needs to read or write are readable or writable, respectively, by "others" — their mode should be 0644 for readable and 0666 for writable. This is because, for security reasons, the HTTP server executes your script as user "nobody", without any special privileges. It can only read (write, execute) files that everybody can read (write, execute). The current directory at execution time is also different (it is usually the server's cgi-bin directory) and the set of environment variables is also different from what you get when you log in. In particular, don't count on the shell's search path for executables (PATH) or the Python module search path (PYTHONPATH) to be set to anything interesting.

If you need to load modules from a directory which is not on Python's default module search path, you can change the path in your script, before importing other modules. For example:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(This way, the directory inserted last will be searched first!)

Instructions for non-Unix systems will vary; check your HTTP server's documentation (it will usually have a section on CGI scripts).

20.2.8 Testing your CGI script

Unfortunately, a CGI script will generally not run when you try it from the command line, and a script that works perfectly from the command line may fail mysteriously when run from the server. There's one reason why you should still test your script from the command line: if it contains a syntax error, the Python interpreter won't execute it at all, and the HTTP server will most likely send a cryptic error to the client.

Assuming your script has no syntax errors, yet it does not work, you have no choice but to read the next section.

20.2.9 Debugging CGI scripts

First of all, check for trivial installation errors — reading the section above on installing your CGI script carefully can save you a lot of time. If you wonder whether you have understood the installation procedure correctly, try installing a copy of this module file (cgi.py) as a CGI script. When invoked as a script, the file will dump its environment and the contents of the form in HTML form. Give it the right mode etc, and send it a request. If it's installed in the standard cgi-bin directory, it should be possible to send it a request by entering a URL into your browser of the form:

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

If this gives an error of type 404, the server cannot find the script – perhaps you need to install it in a different directory. If it gives another error, there's an installation problem that you should fix before trying to go any further. If you get a nicely formatted listing of the environment and form content (in this example, the fields should be listed as "addr" with value "At Home" and "name" with value "Joe Blow"), the cgi.py script has been installed correctly. If you follow the same procedure for your own script, you should now be able to debug it.

The next step could be to call the cgi module's test() function from your script: replace its main code with the single statement

```
cgi.test()
```

This should produce the same results as those gotten from installing the cgi.py file itself.

When an ordinary Python script raises an unhandled exception (for whatever reason: of a typo in a module name, a file that can't be opened, etc.), the Python interpreter prints a nice traceback and exits. While the Python interpreter will still do this when your CGI script raises an exception, most likely the traceback will end up in one of the HTTP server's log files, or be discarded altogether.

Fortunately, once you have managed to get your script to execute some code, you can easily send tracebacks to the Web browser using the cgitb module. If you haven't done so already, just add the lines:

```
import cgitb
cgitb.enable()
```

to the top of your script. Then try running it again; when a problem occurs, you should see a detailed report that will likely make apparent the cause of the crash.

If you suspect that there may be a problem in importing the cgitb module, you can use an even more robust approach (which only uses built-in modules):

```
import sys
sys.stderr = sys.stdout
print "Content-Type: text/plain"
print
...your code here...
```

This relies on the Python interpreter to print the traceback. The content type of the output is set to plain text, which disables all HTML processing. If your script works, the raw HTML will be displayed by your client. If it raises an exception, most likely after the first two lines have been printed, a traceback will be displayed. Because no HTML interpretation is going on, the traceback will be readable.

20.2.10 Common problems and solutions

- Most HTTP servers buffer the output from CGI scripts until the script is completed. This means that it is not possible to display a progress report on the client's display while the script is running.
- Check the installation instructions above.
- Check the HTTP server's log files. (tail -f logfile in a separate window may be useful!)
- Always check a script for syntax errors first, by doing something like python script.py.
- If your script does not have any syntax errors, try adding import cgitb; cgitb.enable() to the top of the script.

- When invoking external programs, make sure they can be found. Usually, this means using absolute path names — PATH is usually not set to a very useful value in a CGI script.
- When reading or writing external files, make sure they can be read or written by the userid under which your CGI script will be running: this is typically the userid under which the web server is running, or some explicitly specified userid for a web server's suexec feature.
- Don't try to give a CGI script a set-uid mode. This doesn't work on most systems, and is a security liability as well.

20.3 cgitb — Traceback manager for CGI scripts

New in version 2.2.

The cgitb module provides a special exception handler for Python scripts. (Its name is a bit misleading. It was originally designed to display extensive traceback information in HTML for CGI scripts. It was later generalized to also display this information in plain text.) After this module is activated, if an uncaught exception occurs, a detailed, formatted report will be displayed. The report includes a traceback showing excerpts of the source code for each level, as well as the values of the arguments and local variables to currently running functions, to help you debug the problem. Optionally, you can save this information to a file instead of sending it to the browser.

To enable this feature, simply add this to the top of your CGI script:

```
import cgitb
cgitb.enable()
```

The options to the enable() function control whether the report is displayed in the browser and whether the report is logged to a file for later analysis.

cgitb.enable([display[, logdir[, context[, format]]]])
: This function causes the cgitb module to take over the interpreter's default handling for exceptions by setting the value of sys.excepthook.

  The optional argument display defaults to 1 and can be set to 0 to suppress sending the traceback to the browser. If the argument logdir is present, the traceback reports are written to files. The value of logdir should be a directory where these files will be placed. The optional argument context is the number of lines of context to display around the current line of source code in the traceback; this defaults to 5. If the optional argument format is "html", the output is formatted as HTML. Any other value forces plain text output. The default value is "html".

cgitb.handler([info])
: This function handles an exception using the default settings (that is, show a report in the browser, but don't log to a file). This can be used when you've caught an exception and want to report it using cgitb. The optional info argument should be a 3-tuple containing an exception type, exception value, and traceback object, exactly like the tuple returned by sys.exc_info(). If the info argument is not supplied, the current exception is obtained from sys.exc_info().

20.4 wsgiref — WSGI Utilities and Reference Implementation

New in version 2.5.

The Web Server Gateway Interface (WSGI) is a standard interface between web server software and web applications written in Python. Having a standard interface makes it easy to use an application that supports WSGI with a number of different web servers.

Only authors of web servers and programming frameworks need to know every detail and corner case of the WSGI design. You don't need to understand every detail of WSGI just to install a WSGI application or to write a web application using an existing framework.

wsgiref is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It provides utilities for manipulating WSGI environment variables and response headers, base classes for implementing WSGI servers, a demo HTTP server that serves WSGI applications, and a validation tool that checks WSGI servers and applications for conformance to the WSGI specification (PEP 333).

See https://wsgi.readthedocs.org/ for more information about WSGI, and links to tutorials and other resources.

20.4.1 wsgiref.util – WSGI environment utilities

This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in PEP 333. All of the functions taking an environ parameter expect a WSGI-compliant dictionary to be supplied; please see PEP 333 for a detailed specification.

wsgiref.util.guess_scheme(environ)
: Return a guess for whether wsgi.url_scheme should be "http" or "https", by checking for a HTTPS environment variable in the environ dictionary. The return value is a string.

  This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a HTTPS variable with a value of "1" "yes", or "on" when a request is received via SSL. So, this function returns "https" if such a value is found, and "http" otherwise.

wsgiref.util.request_uri(environ, include_query=1)
: Return the full request URI, optionally including the query string, using the algorithm found in the "URL Reconstruction" section of PEP 333. If include_query is false, the query string is not included in the resulting URI.

wsgiref.util.application_uri(environ)
: Similar to request_uri(), except that the PATH_INFO and QUERY_STRING variables are ignored. The result is the base URI of the application object addressed by the request.

wsgiref.util.shift_path_info(environ)
: Shift a single name from PATH_INFO to SCRIPT_NAME and return the name. The environ dictionary is modified in-place; use a copy if you need to keep the original PATH_INFO or SCRIPT_NAME intact.

  If there are no remaining path segments in PATH_INFO, None is returned.

  Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at /foo, and the request URI path is /foo/bar/baz, and the WSGI application at /foo calls shift_path_info(), it will receive the string "bar", and the environment will be updated to be suitable for passing to a WSGI application at /foo/bar. That is, SCRIPT_NAME will change from /foo to /foo/bar, and PATH_INFO will change from /bar/baz to /baz.

  When PATH_INFO is just a "/", this routine returns an empty string and appends a trailing slash to SCRIPT_NAME, even though empty path segments are normally ignored, and SCRIPT_NAME doesn't normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in /x from ones ending in /x/ when using this routine to do object traversal.

wsgiref.util.setup_testing_defaults(environ)
Update environ with trivial defaults for testing purposes.

This routine adds various parameters required for WSGI, including HTTP_HOST, SERVER_NAME, SERVER_PORT, REQUEST_METHOD, SCRIPT_NAME, PATH_INFO, and all of the PEP 333-defined wsgi.* variables. It only supplies default values, and does not replace any existing settings for these variables.

This routine is intended to make it easier for unit tests of WSGI servers and applications to set up dummy environments. It should NOT be used by actual WSGI servers or applications, since the data is fake!

Example usage:

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It 's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain')]

    start_response(status, headers)

    ret = ["%s: %s\n" % (key, value)
           for key, value in environ.iteritems()]
    return ret

httpd = make_server('', 8000, simple_app)
print "Serving on port 8000..."
httpd.serve_forever()
```

In addition to the environment functions above, the wsgiref.util module also provides these miscellaneous utilities:

wsgiref.util.is_hop_by_hop(header_name)
Return true if 'header_name' is an HTTP/1.1 "Hop-by-Hop" header, as defined by RFC 2616.

class wsgiref.util.FileWrapper(filelike, blksize=8192)
A wrapper to convert a file-like object to an iterator. The resulting objects support both __getitem__() and __iter__() iteration styles, for compatibility with Python 2.1 and Jython. As the object is iterated over, the optional blksize parameter will be repeatedly passed to the filelike object's read() method to obtain strings to yield. When read() returns an empty string, iteration is ended and is not resumable.

If filelike has a close() method, the returned object will also have a close() method, and it will invoke the filelike object's close() method when called.

Example usage:

```
from StringIO import StringIO
from wsgiref.util import FileWrapper

# We 're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)
```

(continues on next page)

(continued from previous page)

```
for chunk in wrapper:
    print chunk
```

20.4.2 wsgiref.headers – WSGI response header tools

This module provides a single class, Headers, for convenient manipulation of WSGI response headers using a mapping-like interface.

class wsgiref.headers.Headers(headers)

Create a mapping-like object wrapping headers, which must be a list of header name/value tuples as described in PEP 333. Any changes made to the new Headers object will directly update the headers list it was created with.

Headers objects support typical mapping operations including __getitem__(), get(), __setitem__(), setdefault(), __delitem__(), __contains__() and has_key(). For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers' existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, Headers objects do not raise an error when you try to get or delete a key that isn't in the wrapped header list. Getting a nonexistent header just returns None, and deleting a nonexistent header does nothing.

Headers objects also support keys(), values(), and items() methods. The lists returned by keys() and items() can include the same key more than once if there is a multi-valued header. The len() of a Headers object is the same as the length of its items(), which is the same as the length of the wrapped header list. In fact, the items() method just returns a copy of the wrapped header list.

Calling str() on a Headers object returns a formatted string suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the string is terminated with a blank line.

In addition to their mapping interface and formatting features, Headers objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters:

get_all(name)

Return a list of all the values for the named header.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

add_header(name, value, **_params)

Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

name is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or None. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the header value parameters in the form name="value". If it is None, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

The above will add a header that looks like this:

```
Content-Disposition: attachment; filename="bud.gif"
```

20.4.3 wsgiref.simple_server – a simple WSGI HTTP server

This module implements a simple HTTP server (based on BaseHTTPServer) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses PATH_INFO to select which application to invoke for each request. (E.g., using the shift_path_info() function from wsgiref.util.)

wsgiref.simple_server.make_server(host, port, app, server_class=WSGIServer, handler_class=WSGIRequestHandler)

Create a new WSGI server listening on host and port, accepting connections for app. The return value is an instance of the supplied server_class, and will process requests using the specified handler_class. app must be a WSGI application object, as defined by PEP 333.

Example usage:

```
from wsgiref.simple_server import make_server, demo_app

httpd = make_server('', 8000, demo_app)
print "Serving HTTP on port 8000..."

# Respond to requests until process is killed
httpd.serve_forever()

# Alternative: serve one request, then exit
httpd.handle_request()
```

wsgiref.simple_server.demo_app(environ, start_response)

This function is a small but complete WSGI application that returns a text page containing the message "Hello world!" and a list of the key/value pairs provided in the environ parameter. It's useful for verifying that a WSGI server (such as wsgiref.simple_server) is able to run a simple WSGI application correctly.

class wsgiref.simple_server.WSGIServer(server_address, RequestHandlerClass)

Create a WSGIServer instance. server_address should be a (host,port) tuple, and RequestHandlerClass should be the subclass of BaseHTTPServer.BaseHTTPRequestHandler that will be used to process requests.

You do not normally need to call this constructor, as the make_server() function can handle all the details for you.

WSGIServer is a subclass of BaseHTTPServer.HTTPServer, so all of its methods (such as serve_forever() and handle_request()) are available. WSGIServer also provides these WSGI-specific methods:

set_app(application)

Sets the callable application as the WSGI application that will receive requests.

get_app()

Returns the currently-set application callable.

Normally, however, you do not need to use these additional methods, as set_app() is normally called by make_server(), and the get_app() exists mainly for the benefit of request handler instances.

class wsgiref.simple_server.WSGIRequestHandler(request, client_address, server)
Create an HTTP handler for the given request (i.e. a socket), client_address (a (host,port) tuple), and server (WSGIServer instance).

You do not need to create instances of this class directly; they are automatically created as needed by WSGIServer objects. You can, however, subclass this class and supply it as a handler_class to the make_server() function. Some possibly relevant methods for overriding in subclasses:

get_environ()
Returns a dictionary containing the WSGI environment for a request. The default implementation copies the contents of the WSGIServer object's base_environ dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in PEP 333.

get_stderr()
Return the object that should be used as the wsgi.errors stream. The default implementation just returns sys.stderr.

handle()
Process the HTTP request. The default implementation creates a handler instance using a wsgiref.handlers class to implement the actual WSGI application interface.

20.4.4 wsgiref.validate — WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code's conformance using wsgiref.validate. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete PEP 333 compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the paste.lint module from Ian Bicking's "Python Paste" library.

wsgiref.validate.validator(application)
Wrap application and return a new WSGI application object. The returned application will forward all requests to the original application, and will check that both the application and the server invoking it are conforming to the WSGI specification and to RFC 2616.

Any detected nonconformance results in an AssertionError being raised; note, however, that how these errors are handled is server-dependent. For example, wsgiref.simple_server and other servers based on wsgiref.handlers (that don't override the error handling methods to do something else) will simply output a message that an error has occurred, and dump the traceback to sys.stderr or some other error stream.

This wrapper may also generate output using the warnings module to indicate behaviors that are questionable but which may not actually be prohibited by PEP 333. Unless they are suppressed using Python command-line options or the warnings API, any such warnings will be written to sys.stderr (not wsgi.errors, unless they happen to be the same object).

Example usage:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server
```

(continues on next page)

(continued from previous page)

```
# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return "Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)

httpd = make_server('', 8000, validator_app)
print "Listening on port 8000...."
httpd.serve_forever()
```

20.4.5 wsgiref.handlers – server/gateway base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

class wsgiref.handlers.CGIHandler

CGI-based invocation via sys.stdin, sys.stdout, sys.stderr and os.environ. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke CGIHandler().run(app), where app is the WSGI application object you wish to invoke.

This class is a subclass of BaseCGIHandler that sets wsgi.run_once to true, wsgi.multithread to false, and wsgi.multiprocess to true, and always uses sys and os to obtain the necessary CGI streams and environment.

class wsgiref.handlers.BaseCGIHandler(stdin, stdout, stderr, environ, multithread=True, multiprocess=False)

Similar to CGIHandler, but instead of using the sys and os modules, the CGI environment and I/O streams are specified explicitly. The multithread and multiprocess values are used to set the wsgi.multithread and wsgi.multiprocess flags for any applications run by the handler instance.

This class is a subclass of SimpleHandler intended for use with software other than HTTP "origin servers". If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a Status: header to send an HTTP status, you probably want to subclass this instead of SimpleHandler.

class wsgiref.handlers.SimpleHandler(stdin, stdout, stderr, environ, multithread=True, multiprocess=False)

Similar to BaseCGIHandler, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of BaseCGIHandler.

This class is a subclass of BaseHandler. It overrides the __init__(), get_stdin(), get_stderr(), add_cgi_vars(), _write(), and _flush() methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the stdin, stdout, stderr, and environ attributes.

class wsgiref.handlers.BaseHandler

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

BaseHandler instances have only one method intended for external use:

run(app)
Run the specified WSGI application, app.

All of the other BaseHandler methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass:

_write(data)
Buffer the string data for transmission to the client. It's okay if this method actually transmits the data; BaseHandler just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

_flush()
Force buffered data to be transmitted to the client. It's okay if this method is a no-op (i.e., if _write() actually sends the data).

get_stdin()
Return an input stream object suitable for use as the wsgi.input of the request currently being processed.

get_stderr()
Return an output stream object suitable for use as the wsgi.errors of the request currently being processed.

add_cgi_vars()
Insert CGI variables for the current request into the environ attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized BaseHandler subclass.

Attributes and methods for customizing the WSGI environment:

wsgi_multithread
The value to be used for the wsgi.multithread environment variable. It defaults to true in BaseHandler, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_multiprocess
The value to be used for the wsgi.multiprocess environment variable. It defaults to true in BaseHandler, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_run_once
The value to be used for the wsgi.run_once environment variable. It defaults to false in BaseHandler, but CGIHandler sets it to true by default.

os_environ
The default environment variables to be included in every request's WSGI environment. By default, this is a copy of os.environ at the time that wsgiref.handlers was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

server_software
If the origin_server attribute is set, this attribute's value is used to set the default SERVER_SOFTWARE WSGI environment variable, and also to set a default Server: header

in HTTP responses. It is ignored for handlers (such as BaseCGIHandler and CGIHandler) that are not HTTP origin servers.

get_scheme()
: Return the URL scheme being used for the current request. The default implementation uses the guess_scheme() function from wsgiref.util to guess whether the scheme should be "http" or "https", based on the current request's environ variables.

setup_environ()
: Set the environ attribute to a fully-populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the get_stdin(), get_stderr(), and add_cgi_vars() methods and the wsgi_file_wrapper attribute. It also inserts a SERVER_SOFTWARE key if not present, as long as the origin_server attribute is a true value and the server_software attribute is set.

Methods and attributes for customizing exception handling:

log_exception(exc_info)
: Log the exc_info tuple in the server log. exc_info is a (type, value, traceback) tuple. The default implementation simply writes the traceback to the request's wsgi.errors stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

traceback_limit
: The maximum number of frames to include in tracebacks output by the default log_exception() method. If None, all frames are included.

error_output(environ, start_response)
: This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

  This method can access the current error information using sys.exc_info(), and should pass that information to start_response when calling it (as described in the "Error Handling" section of PEP 333).

  The default implementation just uses the error_status, error_headers, and error_body attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

  Note, however, that it's not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn't include any.

error_status
: The HTTP status used for error responses. This should be a status string as defined in PEP 333; it defaults to a 500 code and message.

error_headers
: The HTTP headers used for error responses. This should be a list of WSGI response headers ((name, value) tuples), as described in PEP 333. The default list just sets the content type to text/plain.

error_body
: The error response body. This should be an HTTP response body string. It defaults to the plain text, "A server error occurred. Please contact the administrator."

Methods and attributes for PEP 333's "Optional Platform-Specific File Handling" feature:

wsgi_file_wrapper
: A wsgi.file_wrapper factory, or None. The default value of this attribute is the FileWrapper class from wsgiref.util.

sendfile()
: Override to implement platform-specific file transmission. This method is called only if the application's return value is an instance of the class specified by the wsgi_file_wrapper attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes:

origin_server
: This attribute should be set to a true value if the handler's _write() and _flush() are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special Status: header.

  This attribute's default value is true in BaseHandler, but false in BaseCGIHandler and CGIHandler.

http_version
: If origin_server is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to "1.0".

20.4.6 Examples

This is a working "Hello World" WSGI application:

```
from wsgiref.simple_server import make_server

# Every WSGI application must have an application object - a callable
# object that accepts two arguments. For that purpose, we're going to
# use a function (note that you're not limited to a function, you can
# use a class for example). The first argument passed to the function
# is a dictionary containing CGI-style environment variables and the
# second variable is the callable object (see PEP 333).
def hello_world_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return ["Hello World"]

httpd = make_server('', 8000, hello_world_app)
print "Serving on port 8000..."

# Serve until process is killed
httpd.serve_forever()
```

20.5 urllib — Open arbitrary resources by URL

Note: The urllib module has been split into parts and renamed in Python 3 to urllib.request, urllib.parse, and urllib.error. The 2to3 tool will automatically adapt imports when converting your sources to Python 3. Also note that the urllib.request.urlopen() function in Python 3 is equivalent to urllib2.urlopen() and that urllib.urlopen() has been removed.

This module provides a high-level interface for fetching data across the World Wide Web. In particular, the urlopen() function is similar to the built-in function open(), but accepts Universal Resource Locators (URLs) instead of filenames. Some restrictions apply — it can only open URLs for reading, and no seek operations are available.

See also:

The Requests package is recommended for a higher-level HTTP client interface.

Changed in version 2.7.9: For HTTPS URIs, urllib performs all the neccessary certificate and hostname checks by default.

> Warning: For Python versions earlier than 2.7.9, urllib does not attempt to validate the server certificates of HTTPS URIs. Use at your own risk!

20.5.1 High-level interface

urllib.urlopen(url[, data[, proxies[, context]]])

Open a network object denoted by a URL for reading. If the URL does not have a scheme identifier, or if it has file: as its scheme identifier, this opens a local file (without universal newlines); otherwise it opens a socket to a server somewhere on the network. If the connection cannot be made the IOError exception is raised. If all went well, a file-like object is returned. This supports the following methods: read(), readline(), readlines(), fileno(), close(), info(), getcode() and geturl(). It also has proper support for the iterator protocol. One caveat: the read() method, if the size argument is omitted or negative, may not read until the end of the data stream; there is no good way to determine that the entire stream from a socket has been read in the general case.

Except for the info(), getcode() and geturl() methods, these methods have the same interface as for file objects — see section File Objects in this manual. (It is not a built-in file object, however, so it can't be used at those few places where a true built-in file object is required.)

The info() method returns an instance of the class mimetools.Message containing meta-information associated with the URL. When the method is HTTP, these headers are those returned by the server at the head of the retrieved HTML page (including Content-Length and Content-Type). When the method is FTP, a Content-Length header will be present if (as is now usual) the server passed back a file length in response to the FTP retrieval request. A Content-Type header will be present if the MIME type can be guessed. When the method is local-file, returned headers will include a Date representing the file's last-modified time, a Content-Length giving file size, and a Content-Type containing a guess at the file's type. See also the description of the mimetools module.

The geturl() method returns the real URL of the page. In some cases, the HTTP server redirects a client to another URL. The urlopen() function handles this transparently, but in some cases the caller needs to know which URL the client was redirected to. The geturl() method can be used to get at this redirected URL.

The getcode() method returns the HTTP status code that was sent with the response, or None if the URL is no HTTP URL.

If the url uses the http: scheme identifier, the optional data argument may be given to specify a POST request (normally the request type is GET). The data argument must be in standard application/x-www-form-urlencoded format; see the urlencode() function below.

The urlopen() function works transparently with proxies which do not require authentication. In a Unix or Windows environment, set the http_proxy, or ftp_proxy environment variables to a URL that identifies the proxy server before starting the Python interpreter. For example (the '%' is the command prompt):

```
% http_proxy="http://www.someproxy.com:3128"
% export http_proxy
% python
...
```

The no_proxy environment variable can be used to specify hosts which shouldn't be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with :port appended, for example cern.ch,ncsa.uiuc.edu,some.host:8080.

In a Windows environment, if no proxy environment variables are set, proxy settings are obtained from the registry's Internet Settings section.

In a Mac OS X environment, urlopen() will retrieve proxy information from the OS X System Configuration Framework, which can be managed with Network System Preferences panel.

Alternatively, the optional proxies argument may be used to explicitly specify proxies. It must be a dictionary mapping scheme names to proxy URLs, where an empty dictionary causes no proxies to be used, and None (the default value) causes environmental proxy settings to be used as discussed above. For example:

```
# Use http://www.someproxy.com:3128 for HTTP proxying
proxies = {'http': 'http://www.someproxy.com:3128'}
filehandle = urllib.urlopen(some_url, proxies=proxies)
# Don't use any proxies
filehandle = urllib.urlopen(some_url, proxies={})
# Use proxies from environment - both versions are equivalent
filehandle = urllib.urlopen(some_url, proxies=None)
filehandle = urllib.urlopen(some_url)
```

Proxies which require authentication for use are not currently supported; this is considered an implementation limitation.

The context parameter may be set to a ssl.SSLContext instance to configure the SSL settings that are used if urlopen() makes a HTTPS connection.

Changed in version 2.3: Added the proxies support.

Changed in version 2.6: Added getcode() to returned object and support for the no_proxy environment variable.

Changed in version 2.7.9: The context parameter was added. All the neccessary certificate and hostname checks are done by default.

Deprecated since version 2.6: The urlopen() function has been removed in Python 3 in favor of urllib2.urlopen().

urllib.urlretrieve(url[, filename[, reporthook[, data[, context]]]])

Copy a network object denoted by a URL to a local file, if necessary. If the URL points to a local file, or a valid cached copy of the object exists, the object is not copied. Return a tuple (filename, headers) where filename is the local file name under which the object can be found, and headers is whatever the info() method of the object returned by urlopen() returned (for a remote object, possibly cached). Exceptions are the same as for urlopen().

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be -1 on older FTP servers which do not return a file size in response to a retrieval request.

If the url uses the http: scheme identifier, the optional data argument may be given to specify a POST request (normally the request type is GET). The data argument must in standard application/x-www-form-urlencoded format; see the urlencode() function below.

The context parameter may be set to a ssl.SSLContext instance to configure the SSL settings that are used if urlretrieve() makes a HTTPS connection.

Changed in version 2.5: urlretrieve() will raise ContentTooShortError when it detects that the amount of data available was less than the expected amount (which is the size reported by a Content-Length header). This can occur, for example, when the download is interrupted.

The Content-Length is treated as a lower bound: if there's more data to read, urlretrieve() reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the content attribute of the exception instance.

If no Content-Length header was supplied, urlretrieve() can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

Changed in version 2.7.9: The context parameter was added. All the neccessary certificate and hostname checks are done by default.

urllib._urlopener
The public functions urlopen() and urlretrieve() create an instance of the FancyURLopener class and use it to perform their requested actions. To override this functionality, programmers can create a subclass of URLopener or FancyURLopener, then assign an instance of that class to the urllib._urlopener variable before calling the desired function. For example, applications may want to specify a different User-Agent header than URLopener defines. This can be accomplished with the following code:

```
import urllib

class AppURLopener(urllib.FancyURLopener):
    version = "App/1.7"

urllib._urlopener = AppURLopener()
```

urllib.urlcleanup()
Clear the cache that may have been built up by previous calls to urlretrieve().

20.5.2 Utility functions

urllib.quote(string[, safe])
Replace special characters in string using the %xx escape. Letters, digits, and the characters '_.-' are never quoted. By default, this function is intended for quoting the path section of the URL. The optional safe parameter specifies additional characters that should not be quoted — its default value is '/'.

Example: quote('/~connolly/') yields '/%7econnolly/'.

urllib.quote_plus(string[, safe])
Like quote(), but also replaces spaces by plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in safe. It also does not have safe default to '/'.

urllib.unquote(string)
Replace %xx escapes by their single-character equivalent.

Example: unquote('/%7Econnolly/') yields '/~connolly/'.

urllib.unquote_plus(string)
Like unquote(), but also replaces plus signs by spaces, as required for unquoting HTML form values.

urllib.urlencode(query[, doseq])
Convert a mapping object or a sequence of two-element tuples to a "percent-encoded" string, suitable to pass to urlopen() above as the optional data argument. This is useful to pass a dictionary of form fields to a POST request. The resulting string is a series of key=value pairs separated by '&' characters, where both key and value are quoted using quote_plus() above. When a sequence of two-element tuples is used as the query argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter doseq is evaluates to True, individual key=value pairs separated by '&' are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence. The urlparse module provides the functions parse_qs() and parse_qsl() which are used to parse query strings into Python data structures.

urllib.pathname2url(path)
Convert the pathname path from the local syntax for a path to the form used in the path component of a URL. This does not produce a complete URL. The return value will already be quoted using the quote() function.

urllib.url2pathname(path)
Convert the path component path from a percent-encoded URL to the local syntax for a path. This does not accept a complete URL. This function uses unquote() to decode path.

urllib.getproxies()
This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named <scheme>_proxy, in case insensitive way, for all operating systems first, and when it cannot find it, looks for proxy information from Mac OSX System Configuration for Mac OS X and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.

Note: If the environment variable REQUEST_METHOD is set, which usually indicates your script is running in a CGI environment, the environment variable HTTP_PROXY (uppercase _PROXY) will be ignored. This is because that variable can be injected by a client using the "Proxy:" HTTP header. If you need to use an HTTP proxy in a CGI environment, either use ProxyHandler explicitly, or make sure the variable name is in lowercase (or at least the _proxy suffix).

Note: urllib also exposes certain utility functions like splittype, splithost and others parsing URL into various components. But it is recommended to use urlparse for parsing URLs rather than using these functions directly. Python 3 does not expose these helper functions from urllib.parse module.

20.5.3 URL Opener objects

class urllib.URLopener([proxies[, context[, **x509]]])
Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than http:, ftp:, or file:, you probably want to use FancyURLopener.

By default, the URLopener class sends a User-Agent header of urllib/VVV, where VVV is the urllib version number. Applications can define their own User-Agent header by subclassing URLopener or FancyURLopener and setting the class attribute version to an appropriate string value in the subclass definition.

The optional proxies parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is None, in which case environmental proxy settings will be used if present, as discussed in the definition of urlopen(), above.

The context parameter may be a ssl.SSLContext instance. If given, it defines the SSL settings the opener uses to make HTTPS connections.

Additional keyword parameters, collected in x509, may be used for authentication of the client when using the https: scheme. The keywords key_file and cert_file are supported to provide an SSL key and certificate; both are needed to support client authentication.

URLopener objects will raise an IOError exception if the server returns an error code.

Changed in version 2.7.9: The context parameter was added. All the neccessary certificate and hostname checks are done by default.

open(fullurl[, data])
: Open fullurl using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, open_unknown() is called. The data argument has the same meaning as the data argument of urlopen().

open_unknown(fullurl[, data])
: Overridable interface to open unknown URL types.

retrieve(url[, filename[, reporthook[, data]]])
: Retrieves the contents of url and places it in filename. The return value is a tuple consisting of a local filename and either a mimetools.Message object containing the response headers (for remote URLs) or None (for local URLs). The caller must then open and read the contents of filename. If filename is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and filename is not given, the filename is the output of tempfile.mktemp() with a suffix that matches the suffix of the last path component of the input URL. If reporthook is given, it must be a function accepting three numeric parameters. It will be called after each chunk of data is read from the network. reporthook is ignored for local URLs.

  If the url uses the http: scheme identifier, the optional data argument may be given to specify a POST request (normally the request type is GET). The data argument must in standard application/x-www-form-urlencoded format; see the urlencode() function below.

version
: Variable that specifies the user agent of the opener object. To get urllib to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.

class urllib.FancyURLopener(...)
: FancyURLopener subclasses URLopener providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the Location header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is bounded by the value of the maxtries attribute, which defaults to 10.

  For all other response codes, the method http_error_default() is called which you can override in subclasses to handle the error appropriately.

Note: According to the letter of RFC 2616, 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and urllib reproduces this behaviour.

The parameters to the constructor are the same as those for URLopener.

Note: When performing basic authentication, a FancyURLopener instance calls its prompt_user_passwd() method. The default implementation asks the users for the required information on the controlling terminal. A subclass may override this method to support more appropriate behavior if needed.

The FancyURLopener class offers one additional method that should be overloaded to provide the appropriate behavior:

prompt_user_passwd(host, realm)
Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, (user, password), which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.

exception urllib.ContentTooShortError(msg[, content])
This exception is raised when the urlretrieve() function detects that the amount of the downloaded data is less than the expected amount (given by the Content-Length header). The content attribute stores the downloaded (and supposedly truncated) data.

New in version 2.5.

20.5.4 urllib Restrictions

- Currently, only the following protocols are supported: HTTP, (versions 0.9 and 1.0), FTP, and local files.
- The caching feature of urlretrieve() has been disabled until I find the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The urlopen() and urlretrieve() functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive Web client using these functions without using threads.
- The data returned by urlopen() or urlretrieve() is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the Content-Type header. If the returned data is HTML, you can use the module htmllib to parse it.
- The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a /, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing / has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to

read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the ftplib module, subclassing FancyURLopener, or changing _urlopener to meet your needs.

- This module does not support the use of proxies which require authentication. This may be implemented in the future.
- Although the urllib module contains (undocumented) routines to parse and unparse URL strings, the recommended interface for URL manipulation is in module urlparse.

20.5.5 Examples

Here is an example session that uses the GET method to retrieve a URL containing parameters:

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query?%s" % params)
>>> print f.read()
```

The following example uses the POST method instead:

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query", params)
>>> print f.read()
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```
>>> import urllib
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.FancyURLopener(proxies)
>>> f = opener.open("http://www.python.org")
>>> f.read()
```

The following example uses no proxies at all, overriding environment settings:

```
>>> import urllib
>>> opener = urllib.FancyURLopener({})
>>> f = opener.open("http://www.python.org/")
>>> f.read()
```

20.6 urllib2 — extensible library for opening URLs

Note: The urllib2 module has been split across several modules in Python 3 named urllib.request and urllib.error. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

The urllib2 module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

See also:

The Requests package is recommended for a higher-level HTTP client interface.

The urllib2 module defines the following functions:

urllib2.urlopen(url[, data[, timeout[, cafile[, capath[, cadefault[, context]]]]]])
Open the URL url, which can be either a string or a Request object.

data may be a string specifying additional data to send to the server, or None if no such data is needed. Currently HTTP requests are the only ones that use data; the HTTP request will be a POST instead of a GET when the data parameter is provided. data should be a buffer in the standard application/x-www-form-urlencoded format. The urllib.urlencode() function takes a mapping or sequence of 2-tuples and returns a string in this format. urllib2 module sends HTTP/1.1 requests with Connection:close header included.

The optional timeout parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). This actually only works for HTTP, HTTPS and FTP connections.

If context is specified, it must be a ssl.SSLContext instance describing the various SSL options. See HTTPSConnection for more details.

The optional cafile and capath parameters specify a set of trusted CA certificates for HTTPS requests. cafile should point to a single file containing a bundle of CA certificates, whereas capath should point to a directory of hashed certificate files. More information can be found in ssl.SSLContext.load_verify_locations().

The cadefault parameter is ignored.

This function returns a file-like object with three additional methods:

- geturl() — return the URL of the resource retrieved, commonly used to determine if a redirect was followed
- info() — return the meta-information of the page, such as headers, in the form of an mimetools.Message instance (see Quick Reference to HTTP Headers)
- getcode() — return the HTTP status code of the response.

Raises URLError on errors.

Note that None may be returned if no handler handles the request (though the default installed global OpenerDirector uses UnknownHandler to ensure this never happens).

In addition, if proxy settings are detected (for example, when a *_proxy environment variable like http_proxy is set), ProxyHandler is default installed and makes sure the requests are handled through the proxy.

Changed in version 2.6: timeout was added.

Changed in version 2.7.9: cafile, capath, cadefault, and context were added.

urllib2.install_opener(opener)
Install an OpenerDirector instance as the default global opener. Installing an opener is only necessary if you want urlopen to use that opener; otherwise, simply call OpenerDirector.open() instead of urlopen(). The code does not check for a real OpenerDirector, and any class with the appropriate interface will work.

urllib2.build_opener([handler, ...])
Return an OpenerDirector instance, which chains the handlers in the order given. handlers can be either instances of BaseHandler, or subclasses of BaseHandler (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the handlers, unless the handlers contain them, instances of them or subclasses of them: ProxyHandler (if proxy settings are detected), UnknownHandler, HTTPHandler, HTTPDefaultErrorHandler, HTTPRedirectHandler, FTPHandler, FileHandler, HTTPErrorProcessor.

If the Python installation has SSL support (i.e., if the ssl module can be imported), HTTPSHandler will also be added.

Beginning in Python 2.3, a BaseHandler subclass may also change its handler_order attribute to modify its position in the handlers list.

The following exceptions are raised as appropriate:

exception urllib2.URLError
The handlers raise this exception (or derived exceptions) when they run into a problem. It is a subclass of IOError.

reason
The reason for this error. It can be a message string or another exception instance (socket.error for remote URLs, OSError for local URLs).

exception urllib2.HTTPError
Though being an exception (a subclass of URLError), an HTTPError can also function as a non-exceptional file-like return value (the same thing that urlopen() returns). This is useful when handling exotic HTTP errors, such as requests for authentication.

code
An HTTP status code as defined in RFC 2616. This numeric value corresponds to a value found in the dictionary of codes as found in BaseHTTPServer.BaseHTTPRequestHandler.responses.

reason
The reason for this error. It can be a message string or another exception instance.

The following classes are provided:

class urllib2.Request(url[, data][, headers][, origin_req_host][, unverifiable])
This class is an abstraction of a URL request.

url should be a string containing a valid URL.

data may be a string specifying additional data to send to the server, or None if no such data is needed. Currently HTTP requests are the only ones that use data; the HTTP request will be a POST instead of a GET when the data parameter is provided. data should be a buffer in the standard application/x-www-form-urlencoded format. The urllib.urlencode() function takes a mapping or sequence of 2-tuples and returns a string in this format.

headers should be a dictionary, and will be treated as if add_header() was called with each key and value as arguments. This is often used to "spoof" the User-Agent header value, which is used by a browser to identify itself – some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as "Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11", while urllib2's default user agent string is "Python-urllib/2.6" (on Python 2.6).

The final two arguments are only of interest for correct handling of third-party HTTP cookies:

origin_req_host should be the request-host of the origin transaction, as defined by RFC 2965. It defaults to cookielib.request_host(self). This is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

unverifiable should indicate whether the request is unverifiable, as defined by RFC 2965. It defaults to False. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.

class urllib2.OpenerDirector
: The OpenerDirector class opens URLs via BaseHandlers chained together. It manages the chaining of handlers, and recovery from errors.

class urllib2.BaseHandler
: This is the base class for all registered handlers — and handles only the simple mechanics of registration.

class urllib2.HTTPDefaultErrorHandler
: A class which defines a default handler for HTTP error responses; all responses are turned into HTTPError exceptions.

class urllib2.HTTPRedirectHandler
: A class to handle redirections.

class urllib2.HTTPCookieProcessor([cookiejar])
: A class to handle HTTP Cookies.

class urllib2.ProxyHandler([proxies])
: Cause requests to go through a proxy. If proxies is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables <protocol>_proxy. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry's Internet Settings section, and in a Mac OS X environment proxy information is retrieved from the OS X System Configuration Framework.

  To disable autodetected proxy pass an empty dictionary.

  Note: HTTP_PROXY will be ignored if a variable REQUEST_METHOD is set; see the documentation on getproxies().

  ---

class urllib2.HTTPPasswordMgr
: Keep a database of (realm, uri) -> (user, password) mappings.

class urllib2.HTTPPasswordMgrWithDefaultRealm
: Keep a database of (realm, uri) -> (user, password) mappings. A realm of None is considered a catch-all realm, which is searched if no other realm fits.

class urllib2.AbstractBasicAuthHandler([password_mgr])
: This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib2.HTTPBasicAuthHandler([password_mgr])
: Handle authentication with the remote host. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib2.ProxyBasicAuthHandler([password_mgr])
: Handle authentication with the proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib2.AbstractDigestAuthHandler([password_mgr])
: This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib2.HTTPDigestAuthHandler([password_mgr])
: Handle authentication with the remote host. password_mgr, if given, should be something that is

compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib2.ProxyDigestAuthHandler([password_mgr])
Handle authentication with the proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib2.HTTPHandler
A class to handle opening of HTTP URLs.

class urllib2.HTTPSHandler([debuglevel[, context]])
A class to handle opening of HTTPS URLs. context has the same meaning as for httplib.HTTPSConnection.

Changed in version 2.7.9: context added.

class urllib2.FileHandler
Open local files.

class urllib2.FTPHandler
Open FTP URLs.

class urllib2.CacheFTPHandler
Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class urllib2.UnknownHandler
A catch-all class to handle unknown URLs.

class urllib2.HTTPErrorProcessor
Process HTTP error responses.

20.6.1 Request Objects

The following methods describe all of Request's public interface, and so all must be overridden in subclasses.

Request.add_data(data)
Set the Request data to data. This is ignored by all handlers except HTTP handlers — and there it should be a byte string, and will change the request to be POST rather than GET.

Request.get_method()
Return a string indicating the HTTP request method. This is only meaningful for HTTP requests, and currently always returns 'GET' or 'POST'.

Request.has_data()
Return whether the instance has a non-None data.

Request.get_data()
Return the instance's data.

Request.add_header(key, val)
Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the key collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header.

Request.add_unredirected_header(key, header)
Add a header that will not be added to a redirected request.

New in version 2.4.

Request.has_header(header)
Return whether the instance has the named header (checks both regular and unredirected).

New in version 2.4.

Request.get_full_url()
Return the URL given in the constructor.

Request.get_type()
Return the type of the URL — also known as the scheme.

Request.get_host()
Return the host to which a connection will be made.

Request.get_selector()
Return the selector — the part of the URL that is sent to the server.

Request.get_header(header_name, default=None)
Return the value of the given header. If the header is not present, return the default value.

Request.header_items()
Return a list of tuples (header_name, header_value) of the Request headers.

Request.set_proxy(host, type)
Prepare the request by connecting to a proxy server. The host and type will replace those of the instance, and the instance's selector will be the original URL given in the constructor.

Request.get_origin_req_host()
Return the request-host of the origin transaction, as defined by RFC 2965. See the documentation for the Request constructor.

Request.is_unverifiable()
Return whether the request is unverifiable, as defined by RFC 2965. See the documentation for the Request constructor.

20.6.2 OpenerDirector Objects

OpenerDirector instances have the following methods:

OpenerDirector.add_handler(handler)
handler should be an instance of BaseHandler. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case).

- protocol_open — signal that the handler knows how to open protocol URLs.
- http_error_type — signal that the handler knows how to handle HTTP errors with HTTP error code type.
- protocol_error — signal that the handler knows how to handle errors from (non-http) protocol.
- protocol_request — signal that the handler knows how to pre-process protocol requests.
- protocol_response — signal that the handler knows how to post-process protocol responses.

OpenerDirector.open(url[, data][, timeout])
Open the given url (which can be a request object or a string), optionally passing the given data. Arguments, return values and exceptions raised are the same as those of urlopen() (which simply calls the open() method on the currently installed global OpenerDirector). The optional timeout parameter

specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections).

Changed in version 2.6: timeout was added.

OpenerDirector.error(proto[, arg[, ...]])
Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the http_error_*() methods of the handler classes.

Return values and exceptions raised are the same as those of urlopen().

OpenerDirector objects open URLs in three stages:

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like protocol_request has that method called to pre-process the request.
2. Handlers with a method named like protocol_open are called to handle the request. This stage ends when a handler either returns a non-None value (ie. a response), or raises an exception (usually URLError). Exceptions are allowed to propagate.

   In fact, the above algorithm is first tried for methods named default_open(). If all such methods return None, the algorithm is repeated for methods named like protocol_open. If all such methods return None, the algorithm is repeated for methods named unknown_open().

   Note that the implementation of these methods may involve calls of the parent OpenerDirector instance's open() and error() methods.
3. Every handler with a method named like protocol_response has that method called to post-process the response.

20.6.3 BaseHandler Objects

BaseHandler objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

BaseHandler.add_parent(director)
Add a director as parent.

BaseHandler.close()
Remove any parents.

The following attributes and methods should only be used by classes derived from BaseHandler.

Note: The convention has been adopted that subclasses defining protocol_request() or protocol_response() methods are named *Processor; all others are named *Handler.

BaseHandler.parent
A valid OpenerDirector, which can be used to open using a different protocol, or handle errors.

BaseHandler.default_open(req)
This method is not defined in BaseHandler, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent OpenerDirector. It should return a file-like object as described in the return value of the open() of OpenerDirector, or None. It should raise URLError, unless a truly exceptional thing happens (for example, MemoryError should not be mapped to URLError).

This method will be called before any protocol-specific open method.

BaseHandler.protocol_open(req)
("protocol" is to be replaced by the protocol name.)

This method is not defined in BaseHandler, but subclasses should define it if they want to handle URLs with the given protocol.

This method, if defined, will be called by the parent OpenerDirector. Return values should be the same as for default_open().

BaseHandler.unknown_open(req)
This method is not defined in BaseHandler, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the parent OpenerDirector. Return values should be the same as for default_open().

BaseHandler.http_error_default(req, fp, code, msg, hdrs)
This method is not defined in BaseHandler, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the OpenerDirector getting the error, and should not normally be called in other circumstances.

req will be a Request object, fp will be a file-like object with the HTTP error body, code will be the three-digit code of the error, msg will be the user-visible explanation of the code and hdrs will be a mapping object with the headers of the error.

Return values and exceptions raised should be the same as those of urlopen().

BaseHandler.http_error_nnn(req, fp, code, msg, hdrs)
nnn should be a three-digit HTTP error code. This method is also not defined in BaseHandler, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code nnn occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for http_error_default().

BaseHandler.protocol_request(req)
("protocol" is to be replaced by the protocol name.)

This method is not defined in BaseHandler, but subclasses should define it if they want to pre-process requests of the given protocol.

This method, if defined, will be called by the parent OpenerDirector. req will be a Request object. The return value should be a Request object.

BaseHandler.protocol_response(req, response)
("protocol" is to be replaced by the protocol name.)

This method is not defined in BaseHandler, but subclasses should define it if they want to post-process responses of the given protocol.

This method, if defined, will be called by the parent OpenerDirector. req will be a Request object. response will be an object implementing the same interface as the return value of urlopen(). The return value should implement the same interface as the return value of urlopen().

20.6.4 HTTPRedirectHandler Objects

Note: Some HTTP redirections require action from this module's client code. If this is the case, HTTPError is raised. See RFC 2616 for details of the precise meanings of the various redirection codes.

HTTPRedirectHandler.redirect_request(req, fp, code, msg, hdrs, newurl)
Return a Request or None in response to a redirect. This is called by the default implementations of the http_error_30*() methods when a redirection is received from the server. If a redirection should take place, return a new Request to allow http_error_30*() to perform the redirect to newurl. Otherwise, raise HTTPError if no other handler should try to handle this URL, or return None if you can't but another handler might.

Note: The default implementation of this method does not strictly follow RFC 2616, which says that 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and the default implementation reproduces this behavior.

HTTPRedirectHandler.http_error_301(req, fp, code, msg, hdrs)
Redirect to the Location: or URI: URL. This method is called by the parent OpenerDirector when getting an HTTP 'moved permanently' response.

HTTPRedirectHandler.http_error_302(req, fp, code, msg, hdrs)
The same as http_error_301(), but called for the 'found' response.

HTTPRedirectHandler.http_error_303(req, fp, code, msg, hdrs)
The same as http_error_301(), but called for the 'see other' response.

HTTPRedirectHandler.http_error_307(req, fp, code, msg, hdrs)
The same as http_error_301(), but called for the 'temporary redirect' response.

20.6.5 HTTPCookieProcessor Objects

New in version 2.4.

HTTPCookieProcessor instances have one attribute:

HTTPCookieProcessor.cookiejar
The cookielib.CookieJar in which cookies are stored.

20.6.6 ProxyHandler Objects

ProxyHandler.protocol_open(request)
("protocol" is to be replaced by the protocol name.)

The ProxyHandler will have a method protocol_open for every protocol which has a proxy in the proxies dictionary given in the constructor. The method will modify requests to go through the proxy, by calling request.set_proxy(), and call the next handler in the chain to actually execute the protocol.

20.6.7 HTTPPasswordMgr Objects

These methods are available on HTTPPasswordMgr and HTTPPasswordMgrWithDefaultRealm objects.

HTTPPasswordMgr.add_password(realm, uri, user, passwd)
uri can be either a single URI, or a sequence of URIs. realm, user and passwd must be strings. This causes (user, passwd) to be used as authentication tokens when authentication for realm and a super-URI of any of the given URIs is given.

HTTPPasswordMgr.find_user_password(realm, authuri)
Get user/password for given realm and URI, if any. This method will return (None, None) if there is no matching user/password.

For HTTPPasswordMgrWithDefaultRealm objects, the realm None will be searched if the given realm has no matching user/password.

20.6.8 AbstractBasicAuthHandler Objects

AbstractBasicAuthHandler.http_error_auth_reqed(authreq, host, req, headers)
Handle an authentication request by getting a user/password pair, and re-trying the request. authreq should be the name of the header where the information about the realm is included in the request, host specifies the URL and path to authenticate for, req should be the (failed) Request object, and headers should be the error headers.

host is either an authority (e.g. "python.org") or a URL containing an authority component (e.g. "http://python.org/"). In either case, the authority must not contain a userinfo component (so, "python.org" and "python.org:80" are fine, "joe:password@python.org" is not).

20.6.9 HTTPBasicAuthHandler Objects

HTTPBasicAuthHandler.http_error_401(req, fp, code, msg, hdrs)
Retry the request with authentication information, if available.

20.6.10 ProxyBasicAuthHandler Objects

ProxyBasicAuthHandler.http_error_407(req, fp, code, msg, hdrs)
Retry the request with authentication information, if available.

20.6.11 AbstractDigestAuthHandler Objects

AbstractDigestAuthHandler.http_error_auth_reqed(authreq, host, req, headers)
authreq should be the name of the header where the information about the realm is included in the request, host should be the host to authenticate to, req should be the (failed) Request object, and headers should be the error headers.

20.6.12 HTTPDigestAuthHandler Objects

HTTPDigestAuthHandler.http_error_401(req, fp, code, msg, hdrs)
Retry the request with authentication information, if available.

20.6.13 ProxyDigestAuthHandler Objects

ProxyDigestAuthHandler.http_error_407(req, fp, code, msg, hdrs)
Retry the request with authentication information, if available.

20.6.14 HTTPHandler Objects

HTTPHandler.http_open(req)
Send an HTTP request, which can be either GET or POST, depending on req.has_data().

20.6.15 HTTPSHandler Objects

HTTPSHandler.https_open(req)
Send an HTTPS request, which can be either GET or POST, depending on req.has_data().

20.6.16 FileHandler Objects

FileHandler.file_open(req)
Open the file locally, if there is no host name, or the host name is 'localhost'. Change the protocol to ftp otherwise, and retry opening it using parent.

20.6.17 FTPHandler Objects

FTPHandler.ftp_open(req)
Open the FTP file indicated by req. The login is always done with empty username and password.

20.6.18 CacheFTPHandler Objects

CacheFTPHandler objects are FTPHandler objects with the following additional methods:

CacheFTPHandler.setTimeout(t)
Set timeout of connections to t seconds.

CacheFTPHandler.setMaxConns(m)
Set maximum number of cached connections to m.

20.6.19 UnknownHandler Objects

UnknownHandler.unknown_open()
Raise a URLError exception.

20.6.20 HTTPErrorProcessor Objects

New in version 2.4.

HTTPErrorProcessor.http_response()
Process HTTP error responses.

For 200 error codes, the response object is returned immediately.

For non-200 error codes, this simply passes the job on to the protocol_error_code handler methods, via OpenerDirector.error(). Eventually, urllib2.HTTPDefaultErrorHandler will raise an HTTPError if no other handler handles the error.

HTTPErrorProcessor.https_response()
Process HTTPS error responses.

The behavior is same as http_response().

20.6.21 Examples

In addition to the examples below, more examples are given in urllib-howto.

This example gets the python.org main page and displays the first 100 bytes of it:

```
>>> import urllib2
>>> f = urllib2.urlopen('http://www.python.org/')
>>> print f.read(100)
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<?xml-stylesheet href="./css/ht2html
```

Here we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```
>>> import urllib2
>>> req = urllib2.Request(url='https://localhost/cgi-bin/test.cgi',
...                       data='This data is passed to stdin of the CGI')
>>> f = urllib2.urlopen(req)
>>> print f.read()
Got Data: "This data is passed to stdin of the CGI"
```

The code for the sample CGI used in the above example is:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print 'Content-type: text-plain\n\nGot Data: "%s"' % data
```

Use of Basic HTTP Authentication:

```
import urllib2
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib2.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib2.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib2.install_opener(opener)
urllib2.urlopen('http://www.example.com/login.html')
```

build_opener() provides many handlers by default, including a ProxyHandler. By default, ProxyHandler uses the environment variables named <scheme>_proxy, where <scheme> is the URL scheme involved. For example, the http_proxy environment variable is read to obtain the HTTP proxy's URL.

This example replaces the default ProxyHandler with one that uses programmatically-supplied proxy URLs, and adds proxy authorization support with ProxyBasicAuthHandler.

```
proxy_handler = urllib2.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib2.ProxyBasicAuthHandler()
```

(continues on next page)

(continued from previous page)

```
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib2.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

Adding HTTP headers:

Use the headers argument to the Request constructor, or:

```
import urllib2
req = urllib2.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
r = urllib2.urlopen(req)
```

OpenerDirector automatically adds a User-Agent header to every Request. To change this:

```
import urllib2
opener = urllib2.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

Also, remember that a few standard headers (Content-Length, Content-Type and Host) are added when the Request is passed to urlopen() (or OpenerDirector.open()).

20.7 httplib — HTTP protocol client

Note: The httplib module has been renamed to http.client in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/httplib.py

This module defines classes which implement the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module urllib uses it to handle URLs that use HTTP and HTTPS.

See also:

The Requests package is recommended for a higher-level HTTP client interface.

Note: HTTPS support is only available if the socket module was compiled with SSL support.

Note: The public interface for this module changed substantially in Python 2.0. The HTTP class is retained only for backward compatibility with 1.5.2. It should not be used in new code. Refer to the online docstrings for usage.

The module provides the following classes:

class httplib.HTTPConnection(host[, port[, strict[, timeout[, source_address]]]])
An HTTPConnection instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form host:port, else the default HTTP port (80) is used. When true, the optional parameter strict (which defaults to a false value) causes BadStatusLine to be raised if the status line can't be parsed as a valid HTTP/1.0 or 1.1 status line. If the optional timeout parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional source_address parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from.

For example, the following calls all create instances that connect to the server at the same host and port:

```
>>> h1 = httplib.HTTPConnection('www.cwi.nl')
>>> h2 = httplib.HTTPConnection('www.cwi.nl:80')
>>> h3 = httplib.HTTPConnection('www.cwi.nl', 80)
>>> h3 = httplib.HTTPConnection('www.cwi.nl', 80, timeout=10)
```

New in version 2.0.

Changed in version 2.6: timeout was added.

Changed in version 2.7: source_address was added.

class httplib.HTTPSConnection(host[, port[, key_file[, cert_file[, strict[, timeout[, source_address[, context]]]]]]])
A subclass of HTTPConnection that uses SSL for communication with secure servers. Default port is 443. If context is specified, it must be a ssl.SSLContext instance describing the various SSL options.

key_file and cert_file are deprecated, please use ssl.SSLContext.load_cert_chain() instead, or let ssl.create_default_context() select the system's trusted CA certificates for you.

Please read Security considerations for more information on best practices.

New in version 2.0.

Changed in version 2.6: timeout was added.

Changed in version 2.7: source_address was added.

Changed in version 2.7.9: context was added.

This class now performs all the necessary certificate and hostname checks by default. To revert to the previous, unverified, behavior ssl._create_unverified_context() can be passed to the context parameter.

class httplib.HTTPResponse(sock, debuglevel=0, strict=0)
Class whose instances are returned upon successful connection. Not instantiated directly by user.

New in version 2.0.

class httplib.HTTPMessage
An HTTPMessage instance is used to hold the headers from an HTTP response. It is implemented using the mimetools.Message class and provides utility functions to deal with HTTP Headers. It is not directly instantiated by the users.

The following exceptions are raised as appropriate:

exception httplib.HTTPException
The base class of the other exceptions in this module. It is a subclass of Exception.

New in version 2.0.

exception httplib.NotConnected
A subclass of HTTPException.

New in version 2.0.

exception httplib.InvalidURL
A subclass of HTTPException, raised if a port is given and is either non-numeric or empty.

New in version 2.3.

exception httplib.UnknownProtocol
A subclass of HTTPException.

New in version 2.0.

exception httplib.UnknownTransferEncoding
A subclass of HTTPException.

New in version 2.0.

exception httplib.UnimplementedFileMode
A subclass of HTTPException.

New in version 2.0.

exception httplib.IncompleteRead
A subclass of HTTPException.

New in version 2.0.

exception httplib.ImproperConnectionState
A subclass of HTTPException.

New in version 2.0.

exception httplib.CannotSendRequest
A subclass of ImproperConnectionState.

New in version 2.0.

exception httplib.CannotSendHeader
A subclass of ImproperConnectionState.

New in version 2.0.

exception httplib.ResponseNotReady
A subclass of ImproperConnectionState.

New in version 2.0.

exception httplib.BadStatusLine
A subclass of HTTPException. Raised if a server responds with a HTTP status code that we don't understand.

New in version 2.0.

The constants defined in this module are:

httplib.HTTP_PORT
The default port for the HTTP protocol (always 80).

httplib.HTTPS_PORT
The default port for the HTTPS protocol (always 443).

and also the following constants for integer status codes:

Constant	Value	Definition
CONTINUE	100	HTTP/1.1, RFC 2616, Section 10.1.1
SWITCHING_PROTOCOLS	101	HTTP/1.1, RFC 2616, Section 10.1.2
PROCESSING	102	WEBDAV, RFC 2518, Section 10.1
OK	200	HTTP/1.1, RFC 2616, Section 10.2.1
CREATED	201	HTTP/1.1, RFC 2616, Section 10.2.2
ACCEPTED	202	HTTP/1.1, RFC 2616, Section 10.2.3
NON_AUTHORITATIVE_INFORMATION	203	HTTP/1.1, RFC 2616, Section 10.2.4
NO_CONTENT	204	HTTP/1.1, RFC 2616, Section 10.2.5
RESET_CONTENT	205	HTTP/1.1, RFC 2616, Section 10.2.6
PARTIAL_CONTENT	206	HTTP/1.1, RFC 2616, Section 10.2.7
MULTI_STATUS	207	WEBDAV RFC 2518, Section 10.2
IM_USED	226	Delta encoding in HTTP, RFC 3229, Section 10.4.1
MULTIPLE_CHOICES	300	HTTP/1.1, RFC 2616, Section 10.3.1
MOVED_PERMANENTLY	301	HTTP/1.1, RFC 2616, Section 10.3.2
FOUND	302	HTTP/1.1, RFC 2616, Section 10.3.3
SEE_OTHER	303	HTTP/1.1, RFC 2616, Section 10.3.4
NOT_MODIFIED	304	HTTP/1.1, RFC 2616, Section 10.3.5
USE_PROXY	305	HTTP/1.1, RFC 2616, Section 10.3.6
TEMPORARY_REDIRECT	307	HTTP/1.1, RFC 2616, Section 10.3.8
BAD_REQUEST	400	HTTP/1.1, RFC 2616, Section 10.4.1
UNAUTHORIZED	401	HTTP/1.1, RFC 2616, Section 10.4.2
PAYMENT_REQUIRED	402	HTTP/1.1, RFC 2616, Section 10.4.3
FORBIDDEN	403	HTTP/1.1, RFC 2616, Section 10.4.4
NOT_FOUND	404	HTTP/1.1, RFC 2616, Section 10.4.5
METHOD_NOT_ALLOWED	405	HTTP/1.1, RFC 2616, Section 10.4.6
NOT_ACCEPTABLE	406	HTTP/1.1, RFC 2616, Section 10.4.7
PROXY_AUTHENTICATION_REQUIRED	407	HTTP/1.1, RFC 2616, Section 10.4.8
REQUEST_TIMEOUT	408	HTTP/1.1, RFC 2616, Section 10.4.9
CONFLICT	409	HTTP/1.1, RFC 2616, Section 10.4.10
GONE	410	HTTP/1.1, RFC 2616, Section 10.4.11
LENGTH_REQUIRED	411	HTTP/1.1, RFC 2616, Section 10.4.12
PRECONDITION_FAILED	412	HTTP/1.1, RFC 2616, Section 10.4.13
REQUEST_ENTITY_TOO_LARGE	413	HTTP/1.1, RFC 2616, Section 10.4.14
REQUEST_URI_TOO_LONG	414	HTTP/1.1, RFC 2616, Section 10.4.15
UNSUPPORTED_MEDIA_TYPE	415	HTTP/1.1, RFC 2616, Section 10.4.16
REQUESTED_RANGE_NOT_SATISFIABLE	416	HTTP/1.1, RFC 2616, Section 10.4.17
EXPECTATION_FAILED	417	HTTP/1.1, RFC 2616, Section 10.4.18
UNPROCESSABLE_ENTITY	422	WEBDAV, RFC 2518, Section 10.3
LOCKED	423	WEBDAV RFC 2518, Section 10.4
FAILED_DEPENDENCY	424	WEBDAV, RFC 2518, Section 10.5
UPGRADE_REQUIRED	426	HTTP Upgrade to TLS, RFC 2817, Section 6
INTERNAL_SERVER_ERROR	500	HTTP/1.1, RFC 2616, Section 10.5.1
NOT_IMPLEMENTED	501	HTTP/1.1, RFC 2616, Section 10.5.2
BAD_GATEWAY	502	HTTP/1.1 RFC 2616, Section 10.5.3
SERVICE_UNAVAILABLE	503	HTTP/1.1, RFC 2616, Section 10.5.4
GATEWAY_TIMEOUT	504	HTTP/1.1 RFC 2616, Section 10.5.5
HTTP_VERSION_NOT_SUPPORTED	505	HTTP/1.1, RFC 2616, Section 10.5.6
INSUFFICIENT_STORAGE	507	WEBDAV, RFC 2518, Section 10.6
NOT_EXTENDED	510	An HTTP Extension Framework, RFC 2774, Section 7

httplib.responses

This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example: httplib.responses[httplib.NOT_FOUND] is 'Not Found'.

New in version 2.5.

20.7.1 HTTPConnection Objects

HTTPConnection instances have the following methods:

HTTPConnection.request(method, url[, body[, headers]])
This will send a request to the server using the HTTP request method method and the selector url. If the body argument is present, it should be a string of data to send after the headers are finished. Alternatively, it may be an open file object, in which case the contents of the file is sent; this file object should support fileno() and read() methods. The headers argument should be a mapping of extra HTTP headers to send with the request.

If one is not provided in headers, a Content-Length header is added automatically for all methods if the length of the body can be determined, either from the length of the str representation, or from the reported size of the file on disk. If body is None the header is not set except for methods that expect a body (PUT, POST, and PATCH) in which case it is set to 0.

Changed in version 2.6: body can be a file object.

HTTPConnection.getresponse()
Should be called after a request is sent to get the response from the server. Returns an HTTPResponse instance.

Note: Note that you must have read the whole response before you can send a new request to the server.

HTTPConnection.set_debuglevel(level)
Set the debugging level (the amount of debugging output printed). The default debug level is 0, meaning no debugging output is printed.

HTTPConnection.set_tunnel(host, port=None, headers=None)
Set the host and the port for HTTP Connect Tunnelling. Normally used when it is required to do HTTPS Connection through a proxy server.

The headers argument should be a mapping of extra HTTP headers to send with the CONNECT request.

New in version 2.7.

HTTPConnection.connect()
Connect to the server specified when the object was created.

HTTPConnection.close()
Close the connection to the server.

As an alternative to using the request() method described above, you can also send your request step by step, by using the four functions below.

HTTPConnection.putrequest(request, selector[, skip_host[, skip_accept_encoding]])
This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the request string, the selector string, and the HTTP version (HTTP/1.1). To disable automatic sending of Host: or Accept-Encoding: headers (for example to accept additional content encodings), specify skip_host or skip_accept_encoding with non-False values.

Changed in version 2.4: skip_accept_encoding argument added.

HTTPConnection.putheader(header, argument[, ...])
Send an RFC 822-style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

HTTPConnection.endheaders(message_body=None)
Send a blank line to the server, signalling the end of the headers. The optional message_body argument can be used to pass a message body associated with the request. The message body will be sent in the same packet as the message headers if it is string, otherwise it is sent in a separate packet.

Changed in version 2.7: message_body was added.

HTTPConnection.send(data)
Send data to the server. This should be used directly only after the endheaders() method has been called and before getresponse() is called.

20.7.2 HTTPResponse Objects

HTTPResponse instances have the following methods and attributes:

HTTPResponse.read([amt])
Reads and returns the response body, or up to the next amt bytes.

HTTPResponse.getheader(name[, default])
Get the contents of the header name, or default if there is no matching header.

HTTPResponse.getheaders()
Return a list of (header, value) tuples.

New in version 2.4.

HTTPResponse.fileno()
Returns the fileno of the underlying socket.

HTTPResponse.msg
A mimetools.Message instance containing the response headers.

HTTPResponse.version
HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

HTTPResponse.status
Status code returned by server.

HTTPResponse.reason
Reason phrase returned by server.

20.7.3 Examples

Here is an example session that uses the GET method:

```
>>> import httplib
>>> conn = httplib.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print r1.status, r1.reason
200 OK
```

(continues on next page)

(continued from previous page)

```
>>> data1 = r1.read()
>>> conn.request("GET", "/")
>>> r2 = conn.getresponse()
>>> print r2.status, r2.reason
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the HEAD method. Note that the HEAD method never returns any data.

```
>>> import httplib
>>> conn = httplib.HTTPSConnection("www.python.org")
>>> conn.request("HEAD","/")
>>> res = conn.getresponse()
>>> print res.status, res.reason
200 OK
>>> data = res.read()
>>> print len(data)
0
>>> data == ''
True
```

Here is an example session that shows how to POST requests:

```
>>> import httplib, urllib
>>> params = urllib.urlencode({'@number': 12524, '@type': 'issue', '@action': 'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...           "Accept": "text/plain"}
>>> conn = httplib.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print response.status, response.reason
302 Found
>>> data = response.read()
>>> data
'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/issue12524</a>'
>>> conn.close()
```

Client side HTTP PUT requests are very similar to POST requests. The difference lies only the server side where HTTP server will allow resources to be created via PUT request. Here is an example session that shows how to do PUT request using httplib:

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/foobar
...
>>> import httplib
>>> BODY = "***filecontents***"
>>> conn = httplib.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print response.status, response.reason
200, OK
```

20.8 ftplib — FTP protocol client

Source code: Lib/ftplib.py

This module defines the class FTP and a few related items. The FTP class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other FTP servers. It is also used by the module urllib to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet RFC 959.

Here's a sample session using the ftplib module:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')     # connect to host, default port
>>> ftp.login()                     # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian" directory
>>> ftp.retrlines('LIST')           # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> ftp.retrbinary('RETR README', open('README', 'wb').write)
'226 Transfer complete.'
>>> ftp.quit()
```

The module defines the following items:

class ftplib.FTP([host[, user[, passwd[, acct[, timeout]]]]])
: Return a new instance of the FTP class. When host is given, the method call connect(host) is made. When user is given, additionally the method call login(user, passwd, acct) is made (where passwd and acct default to the empty string when not given). The optional timeout parameter specifies a timeout in seconds for blocking operations like the connection attempt (if is not specified, the global default timeout setting will be used).

  Changed in version 2.6: timeout was added.

class ftplib.FTP_TLS([host[, user[, passwd[, acct[, keyfile[, certfile[, context[, timeout]]]]]]]])
: A FTP subclass which adds TLS support to FTP as described in RFC 4217. Connect as usual to port 21 implicitly securing the FTP control connection before authenticating. Securing the data connection requires the user to explicitly ask for it by calling the prot_p() method. context is a ssl.SSLContext object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read Security considerations for best practices.

  keyfile and certfile are a legacy alternative to context – they can point to PEM-formatted private key and certificate chain files (respectively) for the SSL connection.

  New in version 2.7.

  Changed in version 2.7.10: The context parameter was added.

  Here's a sample session using the FTP_TLS class:

  ```
  >>> from ftplib import FTP_TLS
  >>> ftps = FTP_TLS('ftp.python.org')
  >>> ftps.login()          # login anonymously before securing control channel
  ```

(continues on next page)

(continued from previous page)

```
>>> ftps.prot_p()          # switch to secure data connection
>>> ftps.retrlines('LIST') # list directory content securely
total 9
drwxr-xr-x   8 root     wheel        1024 Jan  3  1994 .
drwxr-xr-x   8 root     wheel        1024 Jan  3  1994 ..
drwxr-xr-x   2 root     wheel        1024 Jan  3  1994 bin
drwxr-xr-x   2 root     wheel        1024 Jan  3  1994 etc
d-wxrwxr-x   2 ftp      wheel        1024 Sep  5 13:43 incoming
drwxr-xr-x   2 root     wheel        1024 Nov 17  1993 lib
drwxr-xr-x   6 1094     wheel        1024 Sep 13 19:07 pub
drwxr-xr-x   3 root     wheel        1024 Jan  3  1994 usr
-rw-r--r--   1 root     root          312 Aug  1  1994 welcome.msg
'226 Transfer complete.'
>>> ftps.quit()
>>>
```

exception ftplib.error_reply
: Exception raised when an unexpected reply is received from the server.

exception ftplib.error_temp
: Exception raised when an error code signifying a temporary error (response codes in the range 400–499) is received.

exception ftplib.error_perm
: Exception raised when an error code signifying a permanent error (response codes in the range 500–599) is received.

exception ftplib.error_proto
: Exception raised when a reply is received from the server that does not fit the response specifications of the File Transfer Protocol, i.e. begin with a digit in the range 1–5.

ftplib.all_errors
: The set of all exceptions (as a tuple) that methods of FTP instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as socket.error and IOError.

See also:

Module netrc
: Parser for the .netrc file format. The file .netrc is typically used by FTP clients to load user authentication information before prompting the user.

The file Tools/scripts/ftpmirror.py in the Python source distribution is a script that can mirror FTP sites, or portions thereof, using the ftplib module. It can be used as an extended example that applies this module.

20.8.1 FTP Objects

Several methods are available in two flavors: one for handling text files and another for binary files. These are named for the command which is used followed by lines for the text version or binary for the binary version.

FTP instances have the following methods:

FTP.set_debuglevel(level)
: Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

FTP.connect(host[, port[, timeout]])
Connect to the given host and port. The default port number is 21, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made.

The optional timeout parameter specifies a timeout in seconds for the connection attempt. If no timeout is passed, the global default timeout setting will be used.

Changed in version 2.6: timeout was added.

FTP.getwelcome()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

FTP.login([user[, passwd[, acct]]])
Log in as the given user. The passwd and acct parameters are optional and default to the empty string. If no user is specified, it defaults to 'anonymous'. If user is 'anonymous', the default passwd is 'anonymous@'. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in. The acct parameter supplies "accounting information"; few systems implement this.

FTP.abort()
Abort a file transfer that is in progress. Using this does not always work, but it's worth a try.

FTP.sendcmd(command)
Send a simple command string to the server and return the response string.

FTP.voidcmd(command)
Send a simple command string to the server and handle the response. Return nothing if a response code corresponding to success (codes in the range 200–299) is received. Raise error_reply otherwise.

FTP.retrbinary(command, callback[, maxblocksize[, rest]])
Retrieve a file in binary transfer mode. command should be an appropriate RETR command: 'RETR filename'. The callback function is called for each block of data received, with a single string argument giving the data block. The optional maxblocksize argument specifies the maximum chunk size to read on the low-level socket object created to do the actual transfer (which will also be the largest size of the data blocks passed to callback). A reasonable default is chosen. rest means the same thing as in the transfercmd() method.

FTP.retrlines(command[, callback])
Retrieve a file or directory listing in ASCII transfer mode. command should be an appropriate RETR command (see retrbinary()) or a command such as LIST, NLST or MLSD (usually just the string 'LIST'). LIST retrieves a list of files and information about those files. NLST retrieves a list of file names. On some servers, MLSD retrieves a machine readable list of files and information about those files. The callback function is called for each line with a string argument containing the line with the trailing CRLF stripped. The default callback prints the line to sys.stdout.

FTP.set_pasv(val)
Enable "passive" mode if val is true, otherwise disable passive mode. (In Python 2.0 and before, passive mode was off by default; in Python 2.1 and later, it is on by default.)

FTP.storbinary(command, fp[, blocksize, callback, rest])
Store a file in binary transfer mode. command should be an appropriate STOR command: "STOR filename". fp is an open file object which is read until EOF using its read() method in blocks of size blocksize to provide the data to be stored. The blocksize argument defaults to 8192. callback is an optional single parameter callable that is called on each block of data after it is sent. rest means the same thing as in the transfercmd() method.

Changed in version 2.1: default for blocksize added.

Changed in version 2.6: callback parameter added.

Changed in version 2.7: rest parameter added.

FTP.storlines(command, fp[, callback])
Store a file in ASCII transfer mode. command should be an appropriate STOR command (see storbinary()). Lines are read until EOF from the open file object fp using its readline() method to provide the data to be stored. callback is an optional single parameter callable that is called on each line after it is sent.

Changed in version 2.6: callback parameter added.

FTP.transfercmd(cmd[, rest])
Initiate a transfer over the data connection. If the transfer is active, send an EPRT or PORT command and the transfer command specified by cmd, and accept the connection. If the server is passive, send an EPSV or PASV command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional rest is given, a REST command is sent to the server, passing rest as an argument. rest is usually a byte offset into the requested file, telling the server to restart sending the file's bytes at the requested offset, skipping over the initial bytes. Note however that RFC 959 requires only that rest be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The transfercmd() method, therefore, converts rest to a string, but no check is performed on the string's contents. If the server does not recognize the REST command, an error_reply exception will be raised. If this happens, simply call transfercmd() without a rest argument.

FTP.ntransfercmd(cmd[, rest])
Like transfercmd(), but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, None will be returned as the expected size. cmd and rest means the same thing as in transfercmd().

FTP.nlst(argument[, ...])
Return a list of file names as returned by the NLST command. The optional argument is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the NLST command.

FTP.dir(argument[, ...])
Produce a directory listing as returned by the LIST command, printing it to standard output. The optional argument is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the LIST command. If the last argument is a function, it is used as a callback function as for retrlines(); the default prints to sys.stdout. This method returns None.

FTP.rename(fromname, toname)
Rename file fromname on the server to toname.

FTP.delete(filename)
Remove the file named filename from the server. If successful, returns the text of the response, otherwise raises error_perm on permission errors or error_reply on other errors.

FTP.cwd(pathname)
Set the current directory on the server.

FTP.mkd(pathname)
Create a new directory on the server.

FTP.pwd()
Return the pathname of the current directory on the server.

FTP.rmd(dirname)
Remove the directory named dirname on the server.

FTP.size(filename)
Request the size of the file named filename on the server. On success, the size of the file is returned as an integer, otherwise None is returned. Note that the SIZE command is not standardized, but is supported by many common server implementations.

FTP.quit()
Send a QUIT command to the server and close the connection. This is the "polite" way to close a connection, but it may raise an exception if the server responds with an error to the QUIT command. This implies a call to the close() method which renders the FTP instance useless for subsequent calls (see below).

FTP.close()
Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to quit(). After this call the FTP instance should not be used any more (after a call to close() or quit() you cannot reopen the connection by issuing another login() method).

20.8.2 FTP_TLS Objects

FTP_TLS class inherits from FTP, defining these additional objects:

FTP_TLS.ssl_version
The SSL version to use (defaults to ssl.PROTOCOL_SSLv23).

FTP_TLS.auth()
Set up secure control connection by using TLS or SSL, depending on what specified in ssl_version() attribute.

FTP_TLS.prot_p()
Set up secure data connection.

FTP_TLS.prot_c()
Set up clear text data connection.

20.9 poplib — POP3 protocol client

Source code: Lib/poplib.py

This module defines a class, POP3, which encapsulates a connection to a POP3 server and implements the protocol as defined in RFC 1725. The POP3 class supports both the minimal and optional command sets. Additionally, this module provides a class POP3_SSL, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the imaplib.IMAP4 class, as IMAP servers tend to be better implemented.

The poplib module provides two classes:

class poplib.POP3(host[, port[, timeout]])
This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If port is omitted, the standard POP3 port (110) is used. The optional timeout parameter

specifies a timeout in seconds for the connection attempt (if not specified, the global default timeout setting will be used).

Changed in version 2.6: timeout was added.

class poplib.POP3_SSL(host[, port[, keyfile[, certfile]]])
This is a subclass of POP3 that connects to the server over an SSL encrypted socket. If port is not specified, 995, the standard POP3-over-SSL port is used. keyfile and certfile are also optional - they can contain a PEM formatted private key and certificate chain file for the SSL connection.

New in version 2.4.

One exception is defined as an attribute of the poplib module:

exception poplib.error_proto
Exception raised on any errors from this module (errors from socket module are not caught). The reason for the exception is passed to the constructor as a string.

See also:

Module imaplib The standard Python IMAP module.

Frequently Asked Questions About Fetchmail The FAQ for the fetchmail POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.

20.9.1 POP3 Objects

All POP3 commands are represented by methods of the same name, in lower-case; most return the response text sent by the server.

An POP3 instance has the following methods:

POP3.set_debuglevel(level)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

POP3.getwelcome()
Returns the greeting string sent by the POP3 server.

POP3.user(username)
Send user command, response should indicate that a password is required.

POP3.pass_(password)
Send password, response includes message count and mailbox size. Note: the mailbox on the server is locked until quit() is called.

POP3.apop(user, secret)
Use the more secure APOP authentication to log into the POP3 server.

POP3.rpop(user)
Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

POP3.stat()
Get mailbox status. The result is a tuple of 2 integers: (message count, mailbox size).

POP3.list([which])
Request message list, result is in the form (response, ['mesg_num octets', ...], octets). If which is set, it is the message to list.

POP3.retr(which)
: Retrieve whole message number which, and set its seen flag. Result is in form (response, ['line', ...], octets).

POP3.dele(which)
: Flag message number which for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

POP3.rset()
: Remove any deletion marks for the mailbox.

POP3.noop()
: Do nothing. Might be used as a keep-alive.

POP3.quit()
: Signoff: commit changes, unlock mailbox, drop connection.

POP3.top(which, howmuch)
: Retrieves the message header plus howmuch lines of the message after the header of message number which. Result is in form (response, ['line', ...], octets).

  The POP3 TOP command this method uses, unlike the RETR command, doesn't set the message's seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

POP3.uidl([which])
: Return message digest (unique id) list. If which is specified, result contains the unique id for that message in the form 'response mesgnum uid, otherwise result is list (response, ['mesgnum uid', ...], octets).

Instances of POP3_SSL have no additional methods. The interface of this subclass is identical to its parent.

20.9.2 POP3 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print j
```

At the end of the module, there is a test section that contains a more extensive example of usage.

20.10 imaplib — IMAP4 protocol client

Source code: Lib/imaplib.py

This module defines three classes, IMAP4, IMAP4_SSL and IMAP4_stream, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in RFC 2060. It is backward compatible with IMAP4 (RFC 1730) servers, but note that the STATUS command is not supported in IMAP4.

Three classes are provided by the imaplib module, IMAP4 is the base class:

class imaplib.IMAP4([host[, port]])
: This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If host is not specified, '' (the local host) is used. If port is omitted, the standard IMAP4 port (143) is used.

Three exceptions are defined as attributes of the IMAP4 class:

exception IMAP4.error
: Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

exception IMAP4.abort
: IMAP4 server errors cause this exception to be raised. This is a sub-class of IMAP4.error. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

exception IMAP4.readonly
: This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of IMAP4.error. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There's also a subclass for secure connections:

class imaplib.IMAP4_SSL([host[, port[, keyfile[, certfile]]]])
: This is a subclass derived from IMAP4 that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If host is not specified, '' (the local host) is used. If port is omitted, the standard IMAP4-over-SSL port (993) is used. keyfile and certfile are also optional - they can contain a PEM formatted private key and certificate chain file for the SSL connection.

The second subclass allows for connections created by a child process:

class imaplib.IMAP4_stream(command)
: This is a subclass derived from IMAP4 that connects to the stdin/stdout file descriptors created by passing command to os.popen2().

  New in version 2.3.

The following utility functions are defined:

imaplib.Internaldate2tuple(datestr)
: Parse an IMAP4 INTERNALDATE string and return corresponding local time. The return value is a time.struct_time instance or None if the string has wrong format.

imaplib.Int2AP(num)
: Converts an integer into a string representation using characters from the set [A .. P].

imaplib.ParseFlags(flagstr)
: Converts an IMAP4 FLAGS response to a tuple of individual flags.

imaplib.Time2Internaldate(date_time)
: Convert date_time to an IMAP4 INTERNALDATE representation. The return value is a string in the form: "DD-Mmm-YYYY HH:MM:SS +HHMM" (including double-quotes). The date_time argument can be a number (int or float) representing seconds since epoch (as returned by time.time()), a 9-tuple representing local time (as returned by time.localtime()), or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an EXPUNGE command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

See also:

Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at the University of Washington's IMAP Information Center (https://www.washington.edu/imap/).

20.10.1 IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either upper-case or lower-case.

All arguments to commands are converted to strings, except for AUTHENTICATE, and the last argument to APPEND which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn't enclosed with either parentheses or double quotes) each string is quoted. However, the password argument to the LOGIN command is always quoted. If you want to avoid having an argument string quoted (eg: the flags argument to STORE) then enclose the string in parentheses (eg: `r'(\Deleted)'`).

Each command returns a tuple: (type, [data, ...]) where type is usually `'OK'` or `'NO'`, and data is either the text from the command response, or mandated results from the command. Each data is either a string, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: 'literal' value).

The message_set options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number (`'1'`), a range of message numbers (`'2:4'`), or a group of non-contiguous ranges separated by commas (`'1:3,6:9'`). A range can contain an asterisk to indicate an infinite upper bound (`'3:*'`).

An IMAP4 instance has the following methods:

IMAP4.append(mailbox, flags, date_time, message)
: Append message to named mailbox.

IMAP4.authenticate(mechanism, authobject)
: Authenticate command — requires response processing.

  mechanism specifies which authentication mechanism is to be used - it should appear in the instance variable capabilities in the form AUTH=mechanism.

  authobject must be a callable object:

  ```
  data = authobject(response)
  ```

  It will be called to process server continuation responses. It should return data that will be encoded and sent to server. It should return None if the client abort response * should be sent instead.

IMAP4.check()
: Checkpoint mailbox on server.

IMAP4.close()
: Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before LOGOUT.

IMAP4.copy(message_set, new_mailbox)
: Copy message_set messages onto end of new_mailbox.

IMAP4.create(mailbox)
: Create new mailbox named mailbox.

IMAP4.delete(mailbox)
Delete old mailbox named mailbox.

IMAP4.deleteacl(mailbox, who)
Delete the ACLs (remove any rights) set for who on mailbox.

New in version 2.4.

IMAP4.expunge()
Permanently remove deleted items from selected mailbox. Generates an EXPUNGE response for each deleted message. Returned data contains a list of EXPUNGE message numbers in order received.

IMAP4.fetch(message_set, message_parts)
Fetch (parts of) messages. message_parts should be a string of message part names enclosed within parentheses, eg: "(UID BODY[TEXT])". Returned data are tuples of message part envelope and data.

IMAP4.getacl(mailbox)
Get the ACLs for mailbox. The method is non-standard, but is supported by the Cyrus server.

IMAP4.getannotation(mailbox, entry, attribute)
Retrieve the specified ANNOTATIONs for mailbox. The method is non-standard, but is supported by the Cyrus server.

New in version 2.5.

IMAP4.getquota(root)
Get the quota root's resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

New in version 2.3.

IMAP4.getquotaroot(mailbox)
Get the list of quota roots for the named mailbox. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

New in version 2.3.

IMAP4.list([directory[, pattern]])
List mailbox names in directory matching pattern. directory defaults to the top-level mail folder, and pattern defaults to match anything. Returned data contains a list of LIST responses.

IMAP4.login(user, password)
Identify the client using a plaintext password. The password will be quoted.

IMAP4.login_cram_md5(user, password)
Force use of CRAM-MD5 authentication when identifying the client to protect the password. Will only work if the server CAPABILITY response includes the phrase AUTH=CRAM-MD5.

New in version 2.3.

IMAP4.logout()
Shutdown connection to server. Returns server BYE response.

IMAP4.lsub([directory[, pattern]])
List subscribed mailbox names in directory matching pattern. directory defaults to the top level directory and pattern defaults to match any mailbox. Returned data are tuples of message part envelope and data.

IMAP4.myrights(mailbox)
Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

New in version 2.4.

IMAP4.namespace()
Returns IMAP namespaces as defined in RFC2342.

New in version 2.3.

IMAP4.noop()
Send NOOP to server.

IMAP4.open(host, port)
Opens socket to port at host. This method is implicitly called by the IMAP4 constructor. The connection objects established by this method will be used in the IMAP4.read(), IMAP4.readline(), IMAP4.send(), and IMAP4.shutdown() methods. You may override this method.

IMAP4.partial(message_num, message_part, start, length)
Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

IMAP4.proxyauth(user)
Assume authentication as user. Allows an authorised administrator to proxy into any user's mailbox.

New in version 2.3.

IMAP4.read(size)
Reads size bytes from the remote server. You may override this method.

IMAP4.readline()
Reads one line from the remote server. You may override this method.

IMAP4.recent()
Prompt server for an update. Returned data is None if no new messages, else value of RECENT response.

IMAP4.rename(oldmailbox, newmailbox)
Rename mailbox named oldmailbox to newmailbox.

IMAP4.response(code)
Return data for response code if received, or None. Returns the given code, instead of the usual type.

IMAP4.search(charset, criterion[, ...])
Search mailbox for matching messages. charset may be None, in which case no CHARSET will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error.

Example:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.select([mailbox[, readonly]])
Select a mailbox. Returned data is the count of messages in mailbox (EXISTS response). The default mailbox is 'INBOX'. If the readonly flag is set, modifications to the mailbox are not allowed.

IMAP4.send(data)
Sends data to the remote server. You may override this method.

IMAP4.setacl(mailbox, who, what)
Set an ACL for mailbox. The method is non-standard, but is supported by the Cyrus server.

IMAP4.setannotation(mailbox, entry, attribute[, ...])
Set ANNOTATIONs for mailbox. The method is non-standard, but is supported by the Cyrus server.

New in version 2.5.

IMAP4.setquota(root, limits)
Set the quota root's resource limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

New in version 2.3.

IMAP4.shutdown()
Close connection established in open. This method is implicitly called by IMAP4.logout(). You may override this method.

IMAP4.socket()
Returns socket instance used to connect to server.

IMAP4.sort(sort_criteria, charset, search_criterion[, ...])
The sort command is a variant of search with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the search_criterion argument(s); a parenthesized list of sort_criteria, and the searching charset. Note that unlike search, the searching charset argument is mandatory. There is also a uid sort command which corresponds to sort the way that uid search corresponds to search. The sort command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an IMAP4rev1 extension command.

IMAP4.status(mailbox, names)
Request named status conditions for mailbox.

IMAP4.store(message_set, command, flag_list)
Alters flag dispositions for messages in mailbox. command is specified by section 6.4.6 of RFC 2060 as being one of "FLAGS", "+FLAGS", or "-FLAGS", optionally with a suffix of ".SILENT".

For example, to set the delete flag on all messages:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

IMAP4.subscribe(mailbox)
Subscribe to new mailbox.

IMAP4.thread(threading_algorithm, charset, search_criterion[, ...])
The thread command is a variant of search with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the search_criterion argument(s); a threading_algorithm, and the searching charset. Note that unlike search, the searching charset argument is mandatory. There is also a uid thread command which corresponds to thread the way that uid search corresponds to search. The thread command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an IMAP4rev1 extension command.

New in version 2.4.

IMAP4.uid(command, arg[, ...])
: Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

IMAP4.unsubscribe(mailbox)
: Unsubscribe from old mailbox.

IMAP4.xatom(name[, arg[, ...]])
: Allow simple extension commands notified by server in CAPABILITY response.

Instances of IMAP4_SSL have just one additional method:

IMAP4_SSL.ssl()
: Returns SSLObject instance used for the secure connection with the server.

The following attributes are defined on instances of IMAP4:

IMAP4.PROTOCOL_VERSION
: The most recent supported protocol in the CAPABILITY response from the server.

IMAP4.debug
: Integer value to control debugging output. The initialize value is taken from the module variable Debug. Values greater than three trace each command.

20.10.2 IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print 'Message %s\n%s\n' % (num, data[0][1])
M.close()
M.logout()
```

20.11 nntplib — NNTP protocol client

Source code: Lib/nntplib.py

This module defines the class NNTP which implements the client side of the NNTP protocol. It can be used to implement a news reader or poster, or automated news processors. For more information on NNTP (Network News Transfer Protocol), see Internet RFC 977.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

```
>>> s = NNTP('news.gmane.org')
>>> resp, count, first, last, name = s.group('gmane.comp.python.committers')
>>> print 'Group', name, 'has', count, 'articles, range', first, 'to', last
Group gmane.comp.python.committers has 1071 articles, range 1 to 1071
>>> resp, subs = s.xhdr('subject', first + '-' + last)
>>> for id, sub in subs[-10:]: print id, sub
...
1062 Re: Mercurial Status?
1063 Re: [python-committers]  (Windows) buildbots on 3.x
1064 Re: Mercurial Status?
1065 Re: Mercurial Status?
1066 Python 2.6.6 status
1067 Commit Privileges for Ask Solem
1068 Re: Commit Privileges for Ask Solem
1069 Re: Commit Privileges for Ask Solem
1070 Re: Commit Privileges for Ask Solem
1071 2.6.6 rc 2
>>> s.quit()
'205 Bye!'
```

To post an article from a file (this assumes that the article has valid headers, and that you have right to post on the particular newsgroup):

```
>>> s = NNTP('news.gmane.org')
>>> f = open('articlefile')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'
```

The module itself defines the following items:

class nntplib.NNTP(host[, port [, user[, password [, readermode] [, usenetrc]]]])
: Return a new instance of the NNTP class, representing a connection to the NNTP server running on host host, listening at port port. The default port is 119. If the optional user and password are provided, or if suitable credentials are present in /.netrc and the optional flag usenetrc is true (the default), the AUTHINFO USER and AUTHINFO PASS commands are used to identify and authenticate the user to the server. If the optional flag readermode is true, then a mode reader command is sent before authentication is performed. Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend to call reader-specific commands, such as group. If you get unexpected NNTPPermanentErrors, you might need to set readermode. readermode defaults to None. usenetrc defaults to True.

  Changed in version 2.4: usenetrc argument added.

exception nntplib.NNTPError
: Derived from the standard exception Exception, this is the base class for all exceptions raised by the nntplib module.

exception nntplib.NNTPReplyError
: Exception raised when an unexpected reply is received from the server. For backwards compatibility, the exception error_reply is equivalent to this class.

exception nntplib.NNTPTemporaryError
: Exception raised when an error code in the range 400–499 is received. For backwards compatibility, the exception error_temp is equivalent to this class.

exception nntplib.NNTPPermanentError
: Exception raised when an error code in the range 500–599 is received. For backwards compatibility,

the exception error_perm is equivalent to this class.

exception nntplib.NNTPProtocolError
Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5. For backwards compatibility, the exception error_proto is equivalent to this class.

exception nntplib.NNTPDataError
Exception raised when there is some error in the response data. For backwards compatibility, the exception error_data is equivalent to this class.

20.11.1 NNTP Objects

NNTP instances have the following methods. The response that is returned as the first item in the return tuple of almost all methods is the server's response: a string beginning with a three-digit code. If the server's response indicates an error, the method raises one of the above exceptions.

NNTP.getwelcome()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

NNTP.set_debuglevel(level)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request or response. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

NNTP.newgroups(date, time[, file])
Send a NEWGROUPS command. The date argument should be a string of the form `'yymmdd'` indicating the date, and time should be a string of the form `'hhmmss'` indicating the time. Return a pair (response, groups) where groups is a list of group names that are new since the given date and time. If the file parameter is supplied, then the output of the NEWGROUPS command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command output. If file is supplied, then the returned list is an empty list.

NNTP.newnews(group, date, time[, file])
Send a NEWNEWS command. Here, group is a group name or `'*'`, and date and time have the same meaning as for newgroups(). Return a pair (response, articles) where articles is a list of message ids. If the file parameter is supplied, then the output of the NEWNEWS command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command output. If file is supplied, then the returned list is an empty list.

NNTP.list([file])
Send a LIST command. Return a pair (response, list) where list is a list of tuples. Each tuple has the form (group, last, first, flag), where group is a group name, last and first are the last and first article numbers (as strings), and flag is `'y'` if posting is allowed, `'n'` if not, and `'m'` if the newsgroup is moderated. (Note the ordering: last, first.) If the file parameter is supplied, then the output of the LIST command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command output. If file is supplied, then the returned list is an empty list.

NNTP.descriptions(grouppattern)
Send a LIST NEWSGROUPS command, where grouppattern is a wildmat string as specified in RFC2980 (it's essentially the same as DOS or UNIX shell wildcard strings). Return a pair (response, list), where list is a list of tuples containing (name, title).

New in version 2.4.

NNTP.description(group)
Get a description for a single group group. If more than one group matches (if 'group' is a real wildmat string), return the first match. If no group matches, return an empty string.

This elides the response code from the server. If the response code is needed, use descriptions().

New in version 2.4.

NNTP.group(name)
Send a GROUP command, where name is the group name. Return a tuple (response, count, first, last, name) where count is the (estimated) number of articles in the group, first is the first article number in the group, last is the last article number in the group, and name is the group name. The numbers are returned as strings.

NNTP.help([file])
Send a HELP command. Return a pair (response, list) where list is a list of help strings. If the file parameter is supplied, then the output of the HELP command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command output. If file is supplied, then the returned list is an empty list.

NNTP.stat(id)
Send a STAT command, where id is the message id (enclosed in '<' and '>') or an article number (as a string). Return a triple (response, number, id) where number is the article number (as a string) and id is the message id (enclosed in '<' and '>').

NNTP.next()
Send a NEXT command. Return as for stat().

NNTP.last()
Send a LAST command. Return as for stat().

NNTP.head(id)
Send a HEAD command, where id has the same meaning as for stat(). Return a tuple (response, number, id, list) where the first three are the same as for stat(), and list is a list of the article's headers (an uninterpreted list of lines, without trailing newlines).

NNTP.body(id[, file])
Send a BODY command, where id has the same meaning as for stat(). If the file parameter is supplied, then the body is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the body. Return as for head(). If file is supplied, then the returned list is an empty list.

NNTP.article(id)
Send an ARTICLE command, where id has the same meaning as for stat(). Return as for head().

NNTP.slave()
Send a SLAVE command. Return the server's response.

NNTP.xhdr(header, string[, file])
Send an XHDR command. This command is not defined in the RFC but is a common extension. The header argument is a header keyword, e.g. 'subject'. The string argument should have the form 'first-last' where first and last are the first and last article numbers to search. Return a pair (response, list), where list is a list of pairs (id, text), where id is an article number (as a string) and text is the text of the requested header for that article. If the file parameter is supplied, then the output of the XHDR command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command output. If file is supplied, then the returned list is an empty list.

NNTP.post(file)
: Post an article using the POST command. The file argument is an open file object which is read until EOF using its readline() method. It should be a well-formed news article, including the required headers. The post() method automatically escapes lines beginning with ..

NNTP.ihave(id, file)
: Send an IHAVE command. id is a message id (enclosed in '<' and '>'). If the response is not an error, treat file exactly as for the post() method.

NNTP.date()
: Return a triple (response, date, time), containing the current date and time in a form suitable for the newnews() and newgroups() methods. This is an optional NNTP extension, and may not be supported by all servers.

NNTP.xgtitle(name[, file])
: Process an XGTITLE command, returning a pair (response, list), where list is a list of tuples containing (name, title). If the file parameter is supplied, then the output of the XGTITLE command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command output. If file is supplied, then the returned list is an empty list. This is an optional NNTP extension, and may not be supported by all servers.

  RFC2980 says "It is suggested that this extension be deprecated". Use descriptions() or description() instead.

NNTP.xover(start, end[, file])
: Return a pair (resp, list). list is a list of tuples, one for each article in the range delimited by the start and end article numbers. Each tuple is of the form (article number, subject, poster, date, id, references, size, lines). If the file parameter is supplied, then the output of the XOVER command is stored in a file. If file is a string, then the method will open a file object with that name, write to it then close it. If file is a file object, then it will start calling write() on it to store the lines of the command output. If file is supplied, then the returned list is an empty list. This is an optional NNTP extension, and may not be supported by all servers.

NNTP.xpath(id)
: Return a pair (resp, path), where path is the directory path to the article with message ID id. This is an optional NNTP extension, and may not be supported by all servers.

NNTP.quit()
: Send a QUIT command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

20.12 smtplib — SMTP protocol client

Source code: Lib/smtplib.py

The smtplib module defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult RFC 821 (Simple Mail Transfer Protocol) and RFC 1869 (SMTP Service Extensions).

class smtplib.SMTP([host[, port[, local_hostname[, timeout]]]])
: An SMTP instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional host and port parameters are given, the SMTP connect() method is called with those parameters during initialization. If specified, local_hostname is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname

is found using socket.getfqdn(). If the connect() call returns anything other than a success code, an SMTPConnectError is raised. The optional timeout parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, socket.timeout is raised.

For normal use, you should only require the initialization/connect, sendmail(), and SMTP.quit() methods. An example is included below.

Changed in version 2.6: timeout was added.

class smtplib.SMTP_SSL([host[, port[, local_hostname[, keyfile[, certfile[, timeout]]]]]])
An SMTP_SSL instance behaves exactly the same as instances of SMTP. SMTP_SSL should be used for situations where SSL is required from the beginning of the connection and using starttls() is not appropriate. If host is not specified, the local host is used. If port is omitted, the standard SMTP-over-SSL port (465) is used. local_hostname has the same meaning as it does for the SMTP class. keyfile and certfile are also optional, and can contain a PEM formatted private key and certificate chain file for the SSL connection. The optional timeout parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, socket.timeout is raised.

New in version 2.6.

class smtplib.LMTP([host[, port[, local_hostname]]])
The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It's common to use Unix sockets for LMTP, so our connect() method must support that as well as a regular host:port server. local_hostname has the same meaning as it does for the SMTP class. To specify a Unix socket, you must use an absolute path for host, starting with a '/'.

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don't support or require any authentication, but your mileage might vary.

New in version 2.6.

A nice selection of exceptions is defined as well:

exception smtplib.SMTPException
The base exception class for all the other exceptions provided by this module.

exception smtplib.SMTPServerDisconnected
This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the SMTP instance before connecting it to a server.

exception smtplib.SMTPResponseException
Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the smtp_code attribute of the error, and the smtp_error attribute is set to the error message.

exception smtplib.SMTPSenderRefused
Sender address refused. In addition to the attributes set by on all SMTPResponseException exceptions, this sets 'sender' to the string that the SMTP server refused.

exception smtplib.SMTPRecipientsRefused
All recipient addresses refused. The errors for each recipient are accessible through the attribute recipients, which is a dictionary of exactly the same sort as SMTP.sendmail() returns.

exception smtplib.SMTPDataError
The SMTP server refused to accept the message data.

exception smtplib.SMTPConnectError
Error occurred during establishment of a connection with the server.

exception smtplib.SMTPHeloError
The server refused our HELO message.

exception smtplib.SMTPAuthenticationError
SMTP authentication went wrong. Most probably the server didn't accept the username/password combination provided.

See also:

RFC 821 - Simple Mail Transfer Protocol Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869 - SMTP Service Extensions Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

20.12.1 SMTP Objects

An SMTP instance has the following methods:

SMTP.set_debuglevel(level)
Set the debug output level. A true value for level results in debug messages for connection and for all messages sent to and received from the server.

SMTP.docmd(cmd[, argstring])
Send a command cmd to the server. The optional argument argstring is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, SMTPServerDisconnected will be raised.

SMTP.connect([host[, port]])
Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (':') followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection response.

SMTP.helo([hostname])
Identify yourself to the SMTP server using HELO. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the helo_resp attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the sendmail() when necessary.

SMTP.ehlo([hostname])
Identify yourself to an ESMTP server using EHLO. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by has_extn(). Also sets several informational attributes: the message returned by the server is stored as the ehlo_resp attribute, does_esmtp is set to true or false depending on whether the server supports ESMTP, and esmtp_features will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use has_extn() before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by sendmail() when necessary.

SMTP.ehlo_or_helo_if_needed()
This method call ehlo() and or helo() if there has been no previous EHLO or HELO command this session. It tries ESMTP EHLO first.

SMTPHeloError The server didn't reply properly to the HELO greeting.

New in version 2.6.

SMTP.has_extn(name)
Return True if name is in the set of SMTP service extensions returned by the server, False otherwise. Case is ignored.

SMTP.verify(address)
Check the validity of an address on this server using SMTP VRFY. Returns a tuple consisting of code 250 and a full RFC 822 address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

Note: Many sites disable SMTP VRFY in order to foil spammers.

SMTP.login(user, password)
Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. This method will return normally if the authentication was successful, or may raise the following exceptions:

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPAuthenticationError The server didn't accept the username/password combination.

SMTPException No suitable authentication method was found.

SMTP.starttls([keyfile[, certfile]])
Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call ehlo() again.

If keyfile and certfile are provided, these are passed to the socket module's ssl() function.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first.

Changed in version 2.6.

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPException The server does not support the STARTTLS extension.

Changed in version 2.6.

RuntimeError SSL/TLS support is not available to your Python interpreter.

SMTP.sendmail(from_addr, to_addrs, msg[, mail_options, rcpt_options])
Send mail. The required arguments are an RFC 822 from-address string, a list of RFC 822 to-address strings (a bare string will be treated as a list with 1 address), and a message string. The caller may pass a list of ESMTP options (such as 8bitmime) to be used in MAIL FROM commands as mail_options. ESMTP options (such as DSN commands) that should be used with all RCPT commands can be passed as rcpt_options. (If you need to use different ESMTP options to different recipients you have to use the low-level methods such as mail(), rcpt() and data() to send the message.)

Note: The from_addr and to_addrs parameters are used to construct the message envelope used by the transport agents. The SMTP does not modify the message headers in any way.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If EHLO fails, HELO will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

This method may raise the following exceptions:

SMTPRecipientsRefused All recipients were refused. Nobody got the mail. The recipients attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPSenderRefused The server didn't accept the from_addr.

SMTPDataError The server replied with an unexpected error code (other than a refusal of a recipient).

Unless otherwise noted, the connection will be open even after an exception is raised.

SMTP.quit()
Terminate the SMTP session and close the connection. Return the result of the SMTP QUIT command.

Changed in version 2.6: Return a value.

Low-level methods corresponding to the standard SMTP/ESMTP commands HELP, RSET, NOOP, MAIL, RCPT, and DATA are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

20.12.2 SMTP Example

This example prompts the user for addresses needed in the message envelope ('To' and 'From' addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn't do any processing of the RFC 822 headers. In particular, the 'To' and 'From' addresses must be included in the message headers explicitly.

```
import smtplib

def prompt(prompt):
    return raw_input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs  = prompt("To: ").split()
print "Enter message, end with ^D (Unix) or ^Z (Windows):"

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
while 1:
```

(continues on next page)

(continued from previous page)

```
    try:
        line = raw_input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print "Message length is " + repr(len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Note: In general, you will want to use the email package's features to construct an email message, which you can then convert to a string and send via sendmail(); see email: Examples.

20.13 smtpd — SMTP Server

Source code: Lib/smtpd.py

This module offers several classes to implement SMTP servers. One is a generic do-nothing implementation, which can be overridden, while the other two offer specific mail-sending strategies.

20.13.1 SMTPServer Objects

class smtpd.SMTPServer(localaddr, remoteaddr)

Create a new SMTPServer object, which binds to local address localaddr. It will treat remoteaddr as an upstream SMTP relayer. Both localaddr and remoteaddr should be a (host, port) tuple. The object inherits from asyncore.dispatcher, and so will insert itself into asyncore's event loop on instantiation.

process_message(peer, mailfrom, rcpttos, data)

Raise NotImplementedError exception. Override this in subclasses to do something useful with this message. Whatever was passed in the constructor as remoteaddr will be available as the _remoteaddr attribute. peer is the remote host's address, mailfrom is the envelope originator, rcpttos are the envelope recipients and data is a string containing the contents of the e-mail (which should be in RFC 2822 format).

20.13.2 DebuggingServer Objects

class smtpd.DebuggingServer(localaddr, remoteaddr)

Create a new debugging server. Arguments are as per SMTPServer. Messages will be discarded, and printed on stdout.

20.13.3 PureProxy Objects

class smtpd.PureProxy(localaddr, remoteaddr)
Create a new pure proxy server. Arguments are as per SMTPServer. Everything will be relayed to remoteaddr. Note that running this has a good chance to make you into an open relay, so please be careful.

20.13.4 MailmanProxy Objects

class smtpd.MailmanProxy(localaddr, remoteaddr)
Create a new pure proxy server. Arguments are as per SMTPServer. Everything will be relayed to remoteaddr, unless local mailman configurations knows about an address, in which case it will be handled via mailman. Note that running this has a good chance to make you into an open relay, so please be careful.

20.14 telnetlib — Telnet client

Source code: Lib/telnetlib.py

The telnetlib module provides a Telnet class that implements the Telnet protocol. See RFC 854 for details about the protocol. In addition, it provides symbolic constants for the protocol characters (see below), and for the telnet options. The symbolic names of the telnet options follow the definitions in arpa/telnet.h, with the leading TELOPT_ removed. For symbolic names of options which are traditionally not included in arpa/telnet.h, see the module source itself.

The symbolic constants for the telnet commands are: IAC, DONT, DO, WONT, WILL, SE (Subnegotiation End), NOP (No Operation), DM (Data Mark), BRK (Break), IP (Interrupt process), AO (Abort output), AYT (Are You There), EC (Erase Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

class telnetlib.Telnet([host[, port[, timeout]]])
Telnet represents a connection to a Telnet server. The instance is initially not connected by default; the open() method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor, to, in which case the connection to the server will be established before the constructor returns. The optional timeout parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not reopen an already connected instance.

This class has many read_*() methods. Note that some of them raise EOFError when the end of the connection is read, because they can return an empty string for other reasons. See the individual descriptions below.

Changed in version 2.6: timeout was added.

See also:

RFC 854 - Telnet Protocol Specification Definition of the Telnet protocol.

20.14.1 Telnet Objects

Telnet instances have the following methods:

Telnet.read_until(expected[, timeout])
Read until a given string, expected, is encountered or until timeout seconds have passed.

When no match is found, return whatever is available instead, possibly the empty string. Raise EOFError if the connection is closed and no cooked data is available.

Telnet.read_all()
Read all data until EOF; block until connection closed.

Telnet.read_some()
Read at least one byte of cooked data unless EOF is hit. Return '' if EOF is hit. Block if no data is immediately available.

Telnet.read_very_eager()
Read everything that can be without blocking in I/O (eager).

Raise EOFError if connection closed and no cooked data available. Return '' if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet.read_eager()
Read readily available data.

Raise EOFError if connection closed and no cooked data available. Return '' if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet.read_lazy()
Process and return data already in the queues (lazy).

Raise EOFError if connection closed and no data available. Return '' if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet.read_very_lazy()
Return any data available in the cooked queue (very lazy).

Raise EOFError if connection closed and no data available. Return '' if no cooked data available otherwise. This method never blocks.

Telnet.read_sb_data()
Return the data collected between a SB/SE pair (suboption begin/end). The callback should access these data when it was invoked with a SE command. This method never blocks.

New in version 2.3.

Telnet.open(host[, port[, timeout]])
Connect to a host. The optional second argument is the port number, which defaults to the standard Telnet port (23). The optional timeout parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not try to reopen an already connected instance.

Changed in version 2.6: timeout was added.

Telnet.msg(msg[, *args])
Print a debug message when the debug level is > 0. If extra arguments are present, they are substituted in the message using the standard string formatting operator.

Telnet.set_debuglevel(debuglevel)
Set the debug level. The higher the value of debuglevel, the more debug output you get (on sys.stdout).

Telnet.close()
Close the connection.

Telnet.get_socket()
Return the socket object used internally.

Telnet.fileno()
Return the file descriptor of the socket object used internally.

Telnet.write(buffer)
Write a string to the socket, doubling any IAC characters. This can block if the connection is blocked. May raise socket.error if the connection is closed.

Telnet.interact()
Interaction function, emulates a very dumb Telnet client.

Telnet.mt_interact()
Multithreaded version of interact().

Telnet.expect(list[, timeout])
Read until one from a list of a regular expressions matches.

The first argument is a list of regular expressions, either compiled (regex objects) or uncompiled (strings). The optional second argument is a timeout, in seconds; the default is to block indefinitely.

Return a tuple of three items: the index in the list of the first regular expression that matches; the match object returned; and the text read up till and including the match.

If end of file is found and no text was read, raise EOFError. Otherwise, when nothing matches, return (-1, None, text) where text is the text received so far (may be the empty string if a timeout happened).

If a regular expression ends with a greedy match (such as .*) or if more than one expression can match the same input, the results are non-deterministic, and may depend on the I/O timing.

Telnet.set_option_negotiation_callback(callback)
Each time a telnet option is read on the input flow, this callback (if set) is called with the following parameters: callback(telnet socket, command (DO/DONT/WILL/WONT), option). No other action is done afterwards by telnetlib.

20.14.2 Telnet Example

A simple example illustrating typical use:

```
import getpass
import sys
import telnetlib

HOST = "localhost"
user = raw_input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until("login: ")
tn.write(user + "\n")
if password:
    tn.read_until("Password: ")
    tn.write(password + "\n")
```

(continues on next page)

(continued from previous page)

```
tn.write("ls\n")
tn.write("exit\n")

print tn.read_all()
```

20.15 uuid — UUID objects according to RFC 4122

New in version 2.5.

This module provides immutable UUID objects (the UUID class) and the functions uuid1(), uuid3(), uuid4(), uuid5() for generating version 1, 3, 4, and 5 UUIDs as specified in RFC 4122.

If all you want is a unique ID, you should probably call uuid1() or uuid4(). Note that uuid1() may compromise privacy since it creates a UUID containing the computer's network address. uuid4() creates a random UUID.

class uuid.UUID([hex[, bytes[, bytes_le[, fields[, int[, version]]]]]])

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes in big-endian order as the bytes argument, a string of 16 bytes in little-endian order as the bytes_le argument, a tuple of six integers (32-bit time_low, 16-bit time_mid, 16-bit time_hi_version, 8-bit clock_seq_hi_variant, 8-bit clock_seq_low, 48-bit node) as the fields argument, or a single 128-bit integer as the int argument. When a string of hex digits is given, curly braces, hyphens, and a URN prefix are all optional. For example, these expressions all yield the same UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes='\x12\x34\x56\x78'*4)
UUID(bytes_le='\x78\x56\x34\x12\x34\x12\x78\x56' +
              '\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

Exactly one of hex, bytes, bytes_le, fields, or int must be given. The version argument is optional; if given, the resulting UUID will have its variant and version number set according to RFC 4122, overriding bits in the given hex, bytes, bytes_le, fields, or int.

UUID instances have these read-only attributes:

UUID.bytes

The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).

UUID.bytes_le

The UUID as a 16-byte string (with time_low, time_mid, and time_hi_version in little-endian byte order).

UUID.fields

A tuple of the six integer fields of the UUID, which are also available as six individual attributes and two derived attributes:

Field	Meaning
time_low	the first 32 bits of the UUID
time_mid	the next 16 bits of the UUID
time_hi_version	the next 16 bits of the UUID
clock_seq_hi_variant	the next 8 bits of the UUID
clock_seq_low	the next 8 bits of the UUID
node	the last 48 bits of the UUID
time	the 60-bit timestamp
clock_seq	the 14-bit sequence number

UUID.hex
: The UUID as a 32-character hexadecimal string.

UUID.int
: The UUID as a 128-bit integer.

UUID.urn
: The UUID as a URN as specified in RFC 4122.

UUID.variant
: The UUID variant, which determines the internal layout of the UUID. This will be one of the constants RESERVED_NCS, RFC_4122, RESERVED_MICROSOFT, or RESERVED_FUTURE.

UUID.version
: The UUID version number (1 through 5, meaningful only when the variant is RFC_4122).

The uuid module defines the following functions:

uuid.getnode()
: Get the hardware address as a 48-bit positive integer. The first time this runs, it may launch a separate program, which could be quite slow. If all attempts to obtain the hardware address fail, we choose a random 48-bit number with its eighth bit set to 1 as recommended in RFC 4122. "Hardware address" means the MAC address of a network interface, and on a machine with multiple network interfaces the MAC address of any one of them may be returned.

uuid.uuid1([node[, clock_seq]])
: Generate a UUID from a host ID, sequence number, and the current time. If node is not given, getnode() is used to obtain the hardware address. If clock_seq is given, it is used as the sequence number; otherwise a random 14-bit sequence number is chosen.

uuid.uuid3(namespace, name)
: Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a string).

uuid.uuid4()
: Generate a random UUID.

uuid.uuid5(namespace, name)
: Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a string).

The uuid module defines the following namespace identifiers for use with uuid3() or uuid5().

uuid.NAMESPACE_DNS
: When this namespace is specified, the name string is a fully-qualified domain name.

uuid.NAMESPACE_URL
: When this namespace is specified, the name string is a URL.

uuid.NAMESPACE_OID
When this namespace is specified, the name string is an ISO OID.

uuid.NAMESPACE_X500
When this namespace is specified, the name string is an X.500 DN in DER or a text output format.

The uuid module defines the following constants for the possible values of the variant attribute:

uuid.RESERVED_NCS
Reserved for NCS compatibility.

uuid.RFC_4122
Specifies the UUID layout given in RFC 4122.

uuid.RESERVED_MICROSOFT
Reserved for Microsoft compatibility.

uuid.RESERVED_FUTURE
Reserved for future definition.

See also:

RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace This specification defines a Uniform Resource Name namespace for UUIDs, the internal format of UUIDs, and methods of generating UUIDs.

20.15.1 Example

Here are some examples of typical usage of the uuid module:

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'
```

(continues on next page)

(continued from previous page)

```
>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

20.16 urlparse — Parse URLs into components

Note: The urlparse module is renamed to urllib.parse in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/urlparse.py

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a "relative URL" to an absolute URL given a "base URL."

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: file, ftp, gopher, hdl, http, https, imap, mailto, mms, news, nntp, prospero, rsync, rtsp, rtspu, sftp, shttp, sip, sips, snews, svn, svn+ssh, telnet, wais.

New in version 2.5: Support for the sftp and sips schemes.

The urlparse module defines the following functions:

urlparse.urlparse(urlstring[, scheme[, allow_fragments]])

Parse a URL into six components, returning a 6-tuple. This corresponds to the general structure of a URL: scheme://netloc/path;parameters?query#fragment. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the path component, which is retained if present. For example:

```
>>> from urlparse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o   # doctest: +NORMALIZE_WHITESPACE
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

Following the syntax specifications in RFC 1808, urlparse recognizes a netloc only if it is properly introduced by '//'. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

```
>>> from urlparse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
```

(continues on next page)

(continued from previous page)

```
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
        params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
        query='', fragment='')
```

If the scheme argument is specified, it gives the default addressing scheme, to be used only if the URL does not specify one. The default value for this argument is the empty string.

If the allow_fragments argument is false, fragment identifiers are not recognized and parsed as part of the preceding component, even if the URL's addressing scheme normally does support them. The default value for this argument is True.

The return value is actually an instance of a subclass of tuple. This class has the following additional read-only convenience attributes:

Attribute	Index	Value	Value if not present
scheme	0	URL scheme specifier	scheme parameter
netloc	1	Network location part	empty string
path	2	Hierarchical path	empty string
params	3	Parameters for last path element	empty string
query	4	Query component	empty string
fragment	5	Fragment identifier	empty string
username		User name	None
password		Password	None
hostname		Host name (lower case)	None
port		Port number as integer, if present	None

See section Results of urlparse() and urlsplit() for more information on the result object.

Changed in version 2.5: Added attributes to return value.

Changed in version 2.7: Added IPv6 URL parsing capabilities.

urlparse.parse_qs(qs[, keep_blank_values[, strict_parsing[, max_num_fields]]])

Parse a query string given as a string argument (data of type application/x-www-form-urlencoded). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument keep_blank_values is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument strict_parsing is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a ValueError exception.

The optional argument max_num_fields is the maximum number of fields to read. If set, then throws a ValueError if there are more than max_num_fields fields read.

Use the urllib.urlencode() function to convert such dictionaries into query strings.

New in version 2.6: Copied from the cgi module.

Changed in version 2.7.16: Added max_num_fields parameter.

urlparse.parse_qsl(qs[, keep_blank_values[, strict_parsing[, max_num_fields]]])
Parse a query string given as a string argument (data of type application/x-www-form-urlencoded). Data are returned as a list of name, value pairs.

The optional argument keep_blank_values is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument strict_parsing is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a ValueError exception.

The optional argument max_num_fields is the maximum number of fields to read. If set, then throws a ValueError if there are more than max_num_fields fields read.

Use the urllib.urlencode() function to convert such lists of pairs into query strings.

New in version 2.6: Copied from the cgi module.

Changed in version 2.7.16: Added max_num_fields parameter.

urlparse.urlunparse(parts)
Construct a URL from a tuple as returned by urlparse(). The parts argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

urlparse.urlsplit(urlstring[, scheme[, allow_fragments]])
This is similar to urlparse(), but does not split the params from the URL. This should generally be used instead of urlparse() if the more recent URL syntax allowing parameters to be applied to each segment of the path portion of the URL (see RFC 2396) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-tuple: (addressing scheme, network location, path, query, fragment identifier).

The return value is actually an instance of a subclass of tuple. This class has the following additional read-only convenience attributes:

Attribute	Index	Value	Value if not present
scheme	0	URL scheme specifier	scheme parameter
netloc	1	Network location part	empty string
path	2	Hierarchical path	empty string
query	3	Query component	empty string
fragment	4	Fragment identifier	empty string
username		User name	None
password		Password	None
hostname		Host name (lower case)	None
port		Port number as integer, if present	None

See section Results of urlparse() and urlsplit() for more information on the result object.

New in version 2.2.

Changed in version 2.5: Added attributes to return value.

urlparse.urlunsplit(parts)
Combine the elements of a tuple as returned by urlsplit() into a complete URL as a string. The parts argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

New in version 2.2.

urlparse.urljoin(base, url[, allow_fragments])
Construct a full ("absolute") URL by combining a "base URL" (base) with another URL (url). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```
>>> from urlparse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The allow_fragments argument has the same meaning and default as for urlparse().

Note: If url is an absolute URL (that is, starting with // or scheme://), the url's host name and/or scheme will be present in the result. For example:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

If you do not want that behavior, preprocess the url with urlsplit() and urlunsplit(), removing possible scheme and netloc parts.

urlparse.urldefrag(url)
If url contains a fragment identifier, returns a modified version of url with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in url, returns url unmodified and an empty string.

See also:

RFC 3986 - Uniform Resource Identifiers This is the current standard (STD66). Any changes to urlparse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

RFC 2732 - Format for Literal IPv6 Addresses in URL's. This specifies the parsing requirements of IPv6 URLs.

RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

RFC 2368 - The mailto URL scheme. Parsing requirements for mailto URL schemes.

RFC 1808 - Relative Uniform Resource Locators This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of "Abnormal Examples" which govern the treatment of border cases.

RFC 1738 - Uniform Resource Locators (URL) This specifies the formal syntax and semantics of absolute URLs.

20.16.1 Results of urlparse() and urlsplit()

The result objects from the urlparse() and urlsplit() functions are subclasses of the tuple type. These subclasses add the attributes described in those functions, as well as provide an additional method:

ParseResult.geturl()
Return the re-combined version of the original URL as a string. This may differ from the original URL

in that the scheme will always be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

The result of this method is a fixpoint if passed back through the original parsing function:

```
>>> import urlparse
>>> url = 'HTTP://www.Python.org/doc/#'
```

```
>>> r1 = urlparse.urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
```

```
>>> r2 = urlparse.urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

New in version 2.5.

The following classes provide the implementations of the parse results:

class urlparse.ParseResult(scheme, netloc, path, params, query, fragment)
Concrete class for urlparse() results.

class urlparse.SplitResult(scheme, netloc, path, query, fragment)
Concrete class for urlsplit() results.

20.17 SocketServer — A framework for network servers

Note: The SocketServer module has been renamed to socketserver in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/SocketServer.py

The SocketServer module simplifies the task of writing network servers.

There are four basic concrete server classes:

class SocketServer.TCPServer(server_address, RequestHandlerClass, bind_and_activate=True)
This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If bind_and_activate is true, the constructor automatically attempts to invoke server_bind() and server_activate(). The other parameters are passed to the BaseServer base class.

class SocketServer.UDPServer(server_address, RequestHandlerClass, bind_and_activate=True)
This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for TCPServer.

class SocketServer.UnixStreamServer(server_address, RequestHandlerClass, bind_and_activate=True)
class SocketServer.UnixDatagramServer(server_address, RequestHandlerClass, bind_and_activate=True)
These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they're not available on non-Unix platforms. The parameters are the same as for TCPServer.

These four classes process requests synchronously; each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the ForkingMixIn and ThreadingMixIn mix-in classes can be used to support asynchronous behaviour.

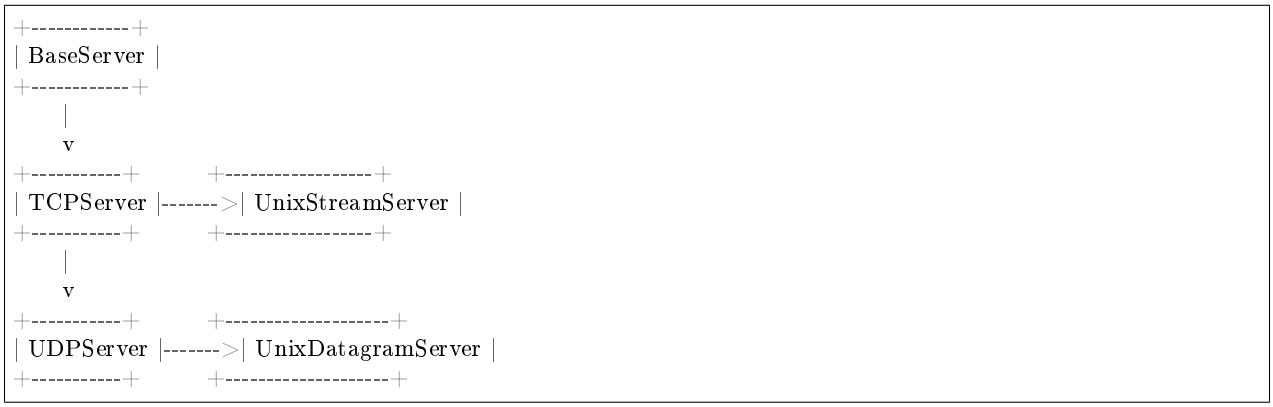
Creating a server requires several steps. First, you must create a request handler class by subclassing the BaseRequestHandler class and overriding its handle() method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class. Then call the handle_request() or serve_forever() method of the server object to process one or many requests. Finally, call server_close() to close the socket.

When inheriting from ThreadingMixIn for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The ThreadingMixIn class defines an attribute daemon_threads, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is False, meaning that Python will not exit until all threads created by ThreadingMixIn have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.

20.17.1 Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:



Note that UnixDatagramServer derives from UDPServer, not from UnixStreamServer — the only difference between an IP and a Unix stream server is the address family, which is simply repeated in both Unix server classes.

class SocketServer.ForkingMixIn
class SocketServer.ThreadingMixIn

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, ThreadingUDPServer is created as follows:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in UDPServer. Setting the various attributes also changes the behavior of the underlying server mechanism.

ForkingMixIn and the Forking classes mentioned below are only available on POSIX platforms that support fork().

class SocketServer.ForkingTCPServer
class SocketServer.ForkingUDPServer

class SocketServer.ThreadingTCPServer
class SocketServer.ThreadingUDPServer
These classes are pre-defined using the mix-in classes.

To implement a service, you must derive a class from BaseRequestHandler and redefine its handle() method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses StreamRequestHandler or DatagramRequestHandler.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service "deaf" while one request is being handled – which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class handle() method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor fork() (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use select() to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used). See asyncore for another way to manage this.

20.17.2 Server Objects

class SocketServer.BaseServer(server_address, RequestHandlerClass)
This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective server_address and RequestHandlerClass attributes.

fileno()
Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to select.select(), to allow monitoring multiple servers in the same process.

handle_request()
Process a single request. This function calls the following methods in order: get_request(), verify_request(), and process_request(). If the user-provided handle() method of the handler class raises an exception, the server's handle_error() method will be called. If no request is received within timeout seconds, handle_timeout() will be called and handle_request() will return.

serve_forever(poll_interval=0.5)
Handle requests until an explicit shutdown() request. Poll for shutdown every poll_interval seconds. Ignores the timeout attribute. If you need to do periodic tasks, do them in another thread.

shutdown()
Tell the serve_forever() loop to stop and wait until it does.

New in version 2.6.

server_close()
Clean up the server. May be overridden.

New in version 2.6.

address_family
The family of protocols to which the server's socket belongs. Common examples are socket.AF_INET and socket.AF_UNIX.

RequestHandlerClass
The user-provided request handler class; an instance of this class is created for each request.

server_address
The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the socket module for details. For Internet protocols, this is a tuple containing a string giving the address, and an integer port number: ('127.0.0.1', 80), for example.

socket
The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:

allow_reuse_address
Whether the server will allow the reuse of an address. This defaults to False, and can be set in subclasses to change the policy.

request_queue_size
The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to request_queue_size requests. Once the queue is full, further requests from clients will get a "Connection denied" error. The default value is usually 5, but this can be overridden by subclasses.

socket_type
The type of socket used by the server; socket.SOCK_STREAM and socket.SOCK_DGRAM are two common values.

timeout
Timeout duration, measured in seconds, or None if no timeout is desired. If handle_request() receives no incoming requests within the timeout period, the handle_timeout() method is called.

There are various server methods that can be overridden by subclasses of base server classes like TCPServer; these methods aren't useful to external users of the server object.

finish_request(request, client_address)
Actually processes the request by instantiating RequestHandlerClass and calling its handle() method.

get_request()
Must accept a request from the socket, and return a 2-tuple containing the new socket object to be used to communicate with the client, and the client's address.

handle_error(request, client_address)
This function is called if the handle() method of a RequestHandlerClass instance raises an exception. The default action is to print the traceback to standard output and continue handling further requests.

handle_timeout()
This function is called when the timeout attribute has been set to a value other than None and the timeout period has passed with no requests being received. The default action for forking

servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

process_request(request, client_address)
: Calls finish_request() to create an instance of the RequestHandlerClass. If desired, this function can create a new process or thread to handle the request; the ForkingMixIn and ThreadingMixIn classes do this.

server_activate()
: Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes listen() on the server's socket. May be overridden.

server_bind()
: Called by the server's constructor to bind the socket to the desired address. May be overridden.

verify_request(request, client_address)
: Must return a Boolean value; if the value is True, the request will be processed, and if it's False, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns True.

20.17.3 Request Handler Objects

class SocketServer.BaseRequestHandler
: This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new handle() method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup()
: Called before the handle() method to perform any initialization actions required. The default implementation does nothing.

handle()
: This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as self.request; the client address as self.client_address; and the server instance as self.server, in case it needs access to per-server information.

  The type of self.request is different for datagram or stream services. For stream services, self.request is a socket object; for datagram services, self.request is a pair of string and socket.

finish()
: Called after the handle() method to perform any clean-up actions required. The default implementation does nothing. If setup() raises an exception, this function will not be called.

class SocketServer.StreamRequestHandler
class SocketServer.DatagramRequestHandler
: These BaseRequestHandler subclasses override the setup() and finish() methods, and provide self.rfile and self.wfile attributes. The self.rfile and self.wfile attributes can be read or written, respectively, to get the request data or return data to the client.

20.17.4 Examples

SocketServer.TCPServer Example

This is the server side:

```
import SocketServer

class MyTCPHandler(SocketServer.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print "{} wrote:".format(self.client_address[0])
        print self.data
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    server = SocketServer.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```
class MyTCPHandler(SocketServer.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print "{} wrote:".format(self.client_address[0])
        print self.data
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the readline() call in the second handler will call recv() multiple times until it encounters a newline character, while the single recv() call in the first handler will just return what has been sent from the client in one sendall() call.

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
```

(continues on next page)

(continued from previous page)

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(data + "\n")

    # Receive data from the server and shut down
    received = sock.recv(1024)
finally:
    sock.close()

print "Sent:     {}".format(data)
print "Received: {}".format(received)
```

The output of the example should look something like this:

Server:

```
$ python TCPServer.py
127.0.0.1 wrote:
hello world with TCP
127.0.0.1 wrote:
python is nice
```

Client:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

SocketServer.UDPServer Example

This is the server side:

```
import SocketServer

class MyUDPHandler(SocketServer.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print "{} wrote:".format(self.client_address[0])
        print data
        socket.sendto(data.upper(), self.client_address)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    server = SocketServer.UDPServer((HOST, PORT), MyUDPHandler)
    server.serve_forever()
```

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(data + "\n", (HOST, PORT))
received = sock.recv(1024)

print "Sent:     {}".format(data)
print "Received: {}".format(received)
```

The output of the example should look exactly like for the TCP server example.

Asynchronous Mixins

To build asynchronous handlers, use the ThreadingMixIn and ForkingMixIn classes.

An example for the ThreadingMixIn class:

```
import socket
import threading
import SocketServer

class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        data = self.request.recv(1024)
        cur_thread = threading.current_thread()
        response = "{}: {}".format(cur_thread.name, data)
        self.request.sendall(response)

class ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass

def client(ip, port, message):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((ip, port))
    try:
        sock.sendall(message)
        response = sock.recv(1024)
        print "Received: {}".format(response)
    finally:
        sock.close()
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    ip, port = server.server_address

    # Start a thread with the server -- that thread will then start one
    # more thread for each request
    server_thread = threading.Thread(target=server.serve_forever)
    # Exit the server thread when the main thread terminates
    server_thread.daemon = True
    server_thread.start()
    print "Server loop running in thread:", server_thread.name

    client(ip, port, "Hello World 1")
    client(ip, port, "Hello World 2")
    client(ip, port, "Hello World 3")

    server.shutdown()
    server.server_close()
```

The output of the example should look something like this:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

The ForkingMixIn class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support fork().

20.18 BaseHTTPServer — Basic HTTP server

Note: The BaseHTTPServer module has been merged into http.server in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/BaseHTTPServer.py

This module defines two classes for implementing HTTP servers (Web servers). Usually, this module isn't used directly, but is used as a basis for building functioning Web servers. See the SimpleHTTPServer and CGIHTTPServer modules.

The first class, HTTPServer, is a SocketServer.TCPServer subclass, and therefore implements the SocketServer.BaseServer interface. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```
def run(server_class=BaseHTTPServer.HTTPServer,
        handler_class=BaseHTTPServer.BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class BaseHTTPServer.HTTPServer(server_address, RequestHandlerClass)
: This class builds on the TCPServer class by storing the server address as instance variables named server_name and server_port. The server is accessible by the handler, typically through the handler's server instance variable.

class BaseHTTPServer.BaseHTTPRequestHandler(request, client_address, server)
: This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). BaseHTTPRequestHandler provides a number of class and instance variables, and methods for use by subclasses.

  The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method SPAM, the do_SPAM() method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the __init__() method.

  BaseHTTPRequestHandler has the following instance variables:

  client_address
  : Contains a tuple of the form (host, port) referring to the client's address.

  server
  : Contains the server instance.

  command
  : Contains the command (request type). For example, 'GET'.

  path
  : Contains the request path.

  request_version
  : Contains the version string from the request. For example, 'HTTP/1.0'.

  headers
  : Holds an instance of the class specified by the MessageClass class variable. This instance parses and manages the headers in the HTTP request.

  rfile
  : Contains an input stream, positioned at the start of the optional input data.

  wfile
  : Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream.

  BaseHTTPRequestHandler has the following class variables:

  server_version
  : Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, 'BaseHTTP/0.2'.

  sys_version
  : Contains the Python system version, in a form usable by the version_string method and the server_version class variable. For example, 'Python/1.4'.

error_message_format
: Specifies a format string for building an error response to the client. It uses parenthesized, keyed format specifiers, so the format operand must be a dictionary. The code key should be an integer, specifying the numeric HTTP error code value. message should be a string containing a (detailed) error message of what occurred, and explain should be an explanation of the error code number. Default message and explain values can found in the responses class variable.

error_content_type
: Specifies the Content-Type HTTP header of error responses sent to the client. The default value is `'text/html'`.

  New in version 2.6: Previously, the content type was always `'text/html'`.

protocol_version
: This specifies the HTTP protocol version used in responses. If set to `'HTTP/1.1'`, the server will permit HTTP persistent connections; however, your server must then include an accurate Content-Length header (using send_header()) in all of its responses to clients. For backwards compatibility, the setting defaults to `'HTTP/1.0'`.

MessageClass
: Specifies a rfc822.Message-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to mimetools.Message.

responses
: This variable contains a mapping of error code integers to two-element tuples containing a short and long message. For example, {code: (shortmessage, longmessage)}. The shortmessage is usually used as the message key in an error response, and longmessage as the explain key (see the error_message_format class variable).

A BaseHTTPRequestHandler instance has the following methods:

handle()
: Calls handle_one_request() once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate do_*() methods.

handle_one_request()
: This method will parse and dispatch the request to the appropriate do_*() method. You should never need to override it.

send_error(code[, message])
: Sends and logs a complete error reply to the client. The numeric code specifies the HTTP error code, with message as optional, more specific text. A complete set of headers is sent, followed by text composed using the error_message_format class variable. The body will be empty if the method is HEAD or the response code is one of the following: 1xx, 204 No Content, 205 Reset Content, 304 Not Modified.

send_response(code[, message])
: Sends a response header and logs the accepted request. The HTTP response line is sent, followed by Server and Date headers. The values for these two headers are picked up from the version_string() and date_time_string() methods, respectively.

send_header(keyword, value)
: Writes a specific HTTP header to the output stream. keyword should specify the header keyword, with value specifying its value.

end_headers()
: Sends a blank line, indicating the end of the HTTP headers in the response.

log_request([code[, size]])
: Logs an accepted (successful) request. code should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the size parameter.

log_error(...)
: Logs an error when a request cannot be fulfilled. By default, it passes the message to log_message(), so it takes the same arguments (format and additional values).

log_message(format, ...)
: Logs an arbitrary message to sys.stderr. This is typically overridden to create custom error logging mechanisms. The format argument is a standard printf-style format string, where the additional arguments to log_message() are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

version_string()
: Returns the server software's version string. This is a combination of the server_version and sys_version class variables.

date_time_string([timestamp])
: Returns the date and time given by timestamp (which must be in the format returned by time.time()), formatted for a message header. If timestamp is omitted, it uses the current date and time.

  The result looks like 'Sun, 06 Nov 1994 08:49:37 GMT'.

  New in version 2.5: The timestamp parameter.

log_date_time_string()
: Returns the current date and time, formatted for logging.

address_string()
: Returns the client address, formatted for logging. A name lookup is performed on the client's IP address.

20.18.1 More examples

To create a server that doesn't run forever, but until some condition is fulfilled:

```
def run_while_true(server_class=BaseHTTPServer.HTTPServer,
                   handler_class=BaseHTTPServer.BaseHTTPRequestHandler):
    """
    This assumes that keep_running() is a function of no arguments which
    is tested initially and after each request.  If its return value
    is true, the server continues.
    """
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    while keep_running():
        httpd.handle_request()
```

See also:

Module CGIHTTPServer Extended request handler that supports CGI scripts.

Module SimpleHTTPServer Basic request handler that limits response to files actually under the document root.

20.19 SimpleHTTPServer — Simple HTTP request handler

Note: The SimpleHTTPServer module has been merged into http.server in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

Warning: mod:SimpleHTTServer is not recommended for production. It only implements basic security checks.

The SimpleHTTPServer module defines a single class, SimpleHTTPRequestHandler, which is interface-compatible with BaseHTTPServer.BaseHTTPRequestHandler.

The SimpleHTTPServer module defines the following class:

class SimpleHTTPServer.SimpleHTTPRequestHandler(request, client_address, server)
This class serves files from the current directory and below, directly mapping the directory structure to HTTP requests.

A lot of the work, such as parsing the request, is done by the base class BaseHTTPServer.BaseHTTPRequestHandler. This class implements the do_GET() and do_HEAD() functions.

The following are defined as class-level attributes of SimpleHTTPRequestHandler:

server_version

This will be "SimpleHTTP/" + __version__, where __version__ is defined at the module level.

extensions_map
A dictionary mapping suffixes into MIME types. The default is signified by an empty string, and is considered to be application/octet-stream. The mapping is used case-insensitively, and so should contain only lower-cased keys.

The SimpleHTTPRequestHandler class defines the following methods:

do_HEAD()
This method serves the 'HEAD' request type: it sends the headers it would send for the equivalent GET request. See the do_GET() method for a more complete explanation of the possible headers.

do_GET()
The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named index.html or index.htm (in that order). If found, the file's contents are returned; otherwise a directory listing is generated by calling the list_directory() method. This method uses os.listdir() to scan the directory, and returns a 404 error response if the listdir() fails.

If the request was mapped to a file, it is opened and the contents are returned. Any IOError exception in opening the requested file is mapped to a 404, 'File not found' error. Otherwise, the content type is guessed by calling the guess_type() method, which in turn uses the extensions_map variable.

A 'Content-type:' header with the guessed content type is output, followed by a 'Content-Length:' header with the file's size and a 'Last-Modified:' header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with text/ the file is opened in text mode; otherwise binary mode is used.

The test() function in the SimpleHTTPServer module is an example which creates a server using the SimpleHTTPRequestHandler as the Handler.

New in version 2.5: The 'Last-Modified' header.

The SimpleHTTPServer module can be used in the following manner in order to set up a very basic web server serving files relative to the current directory.

```
import SimpleHTTPServer
import SocketServer

PORT = 8000

Handler = SimpleHTTPServer.SimpleHTTPRequestHandler

httpd = SocketServer.TCPServer(("", PORT), Handler)

print "serving at port", PORT
httpd.serve_forever()
```

The SimpleHTTPServer module can also be invoked directly using the -m switch of the interpreter with a port number argument. Similar to the previous example, this serves the files relative to the current directory.

```
python -m SimpleHTTPServer 8000
```

See also:

Module BaseHTTPServer Base class implementation for Web server and request handler.

20.20 CGIHTTPServer — CGI-capable HTTP request handler

Note: The CGIHTTPServer module has been merged into http.server in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

The CGIHTTPServer module defines a request-handler class, interface compatible with BaseHTTPServer.BaseHTTPRequestHandler and inherits behavior from SimpleHTTPServer.SimpleHTTPRequestHandler but can also run CGI scripts.

Note: This module can run CGI scripts on Unix and Windows systems.

Note: CGI scripts run by the CGIHTTPRequestHandler class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts the status code.

The CGIHTTPServer module defines the following class:

class CGIHTTPServer.CGIHTTPRequestHandler(request, client_address, server)
This class is used to serve either files or output of CGI scripts from the current directory and be-

low. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in SimpleHTTPServer.SimpleHTTPRequestHandler.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used — the other common server configuration is to treat special extensions as denoting CGI scripts.

The do_GET() and do_HEAD() functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the cgi_directories path.

The CGIHTTPRequestHandler defines the following data member:

cgi_directories
: This defaults to ['/cgi-bin', '/htbin'] and describes directories to treat as containing CGI scripts.

The CGIHTTPRequestHandler defines the following methods:

do_POST()
: This method serves the 'POST' request type, only allowed for CGI scripts. Error 501, "Can only POST to CGI scripts", is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

For example usage, see the implementation of the test() function.

See also:

Module BaseHTTPServer Base class implementation for Web server and request handler.

20.21 cookielib — Cookie handling for HTTP clients

Note: The cookielib module has been renamed to http.cookiejar in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

New in version 2.4.

Source code: Lib/cookielib.py

The cookielib module defines classes for automatic handling of HTTP cookies. It is useful for accessing web sites that require small pieces of data – cookies – to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by RFC 2965 are handled. RFC 2965 handling is switched off by default. RFC 2109 cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the 'policy' in effect. Note that the great majority of cookies on the Internet are Netscape cookies. cookielib attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the max-age and port cookie-attributes introduced with RFC 2965.

Note: The various named parameters found in Set-Cookie and Set-Cookie2 headers (eg. domain and expires) are conventionally referred to as attributes. To distinguish them from Python attributes, the documentation for this module uses the term cookie-attribute instead.

The module defines the following exception:

exception cookielib.LoadError
Instances of FileCookieJar raise this exception on failure to load cookies from a file.

Note: For backwards-compatibility with Python 2.4 (which raised an IOError), LoadError is a subclass of IOError.

The following classes are provided:

class cookielib.CookieJar(policy=None)
policy is an object implementing the CookiePolicy interface.

The CookieJar class stores HTTP cookies. It extracts cookies from HTTP requests, and returns them in HTTP responses. CookieJar instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.

class cookielib.FileCookieJar(filename, delayload=None, policy=None)
policy is an object implementing the CookiePolicy interface. For the other arguments, see the documentation for the corresponding attributes.

A CookieJar which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are NOT loaded from the named file until either the load() or revert() method is called. Subclasses of this class are documented in section FileCookieJar subclasses and co-operation with web browsers.

class cookielib.CookiePolicy
This class is responsible for deciding whether each cookie should be accepted from / returned to the server.

class cookielib.DefaultCookiePolicy(blocked_domains=None, allowed_domains=None, netscape=True, rfc2965=False, rfc2109_as_netscape=None, hide_cookie2=False, strict_domain=False, strict_rfc2965_unverifiable=True, strict_ns_unverifiable=False, strict_ns_domain=DefaultCookiePolicy.DomainLiberal, strict_ns_set_initial_dollar=False, strict_ns_set_path=False)
Constructor arguments should be passed as keyword arguments only. blocked_domains is a sequence of domain names that we never accept cookies from, nor return cookies to. allowed_domains if not None, this is a sequence of the only domains for which we accept and return cookies. For all other arguments, see the documentation for CookiePolicy and DefaultCookiePolicy objects.

DefaultCookiePolicy implements the standard accept / reject rules for Netscape and RFC 2965 cookies. By default, RFC 2109 cookies (ie. cookies received in a Set-Cookie header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or rfc2109_as_netscape is True, RFC 2109 cookies are 'downgraded' by the CookieJar instance to Netscape cookies, by setting the version attribute of the Cookie instance to 0. DefaultCookiePolicy also provides some parameters to allow some fine-tuning of policy.

class cookielib.Cookie
This class represents Netscape, RFC 2109 and RFC 2965 cookies. It is not expected that users of cookielib construct their own Cookie instances. Instead, if necessary, call make_cookies() on a CookieJar instance.

See also:

Module urllib2 URL opening with automatic cookie handling.

Module Cookie HTTP cookie classes, principally useful for server-side code. The cookielib and Cookie modules do not depend on each other.

https://curl.haxx.se/rfc/cookie_spec.html The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the 'Netscape cookie protocol' implemented by all the major

browsers (and cookielib) only bears a passing resemblance to the one sketched out in cookie_spec.html.

RFC 2109 - HTTP State Management Mechanism Obsoleted by RFC 2965. Uses Set-Cookie with version=1.

RFC 2965 - HTTP State Management Mechanism The Netscape protocol with the bugs fixed. Uses Set-Cookie2 in place of Set-Cookie. Not widely used.

http://kristol.org/cookie/errata.html Unfinished errata to RFC 2965.

RFC 2964 - Use of HTTP State Management

20.21.1 CookieJar and FileCookieJar Objects

CookieJar objects support the iterator protocol for iterating over contained Cookie objects.

CookieJar has the following methods:

CookieJar.add_cookie_header(request)
Add correct Cookie header to request.

If policy allows (ie. the rfc2965 and hide_cookie2 attributes of the CookieJar's CookiePolicy instance are true and false respectively), the Cookie2 header is also added when appropriate.

The request object (usually a urllib2.Request instance) must support the methods get_full_url(), get_host(), get_type(), unverifiable(), get_origin_req_host(), has_header(), get_header(), header_items(), and add_unredirected_header(),as documented by urllib2.

CookieJar.extract_cookies(response, request)
Extract cookies from HTTP response and store them in the CookieJar, where allowed by policy.

The CookieJar will look for allowable Set-Cookie and Set-Cookie2 headers in the response argument, and store cookies as appropriate (subject to the CookiePolicy.set_ok() method's approval).

The response object (usually the result of a call to urllib2.urlopen(), or similar) should support an info() method, which returns an object with a getallmatchingheaders() method (usually a mimetools.Message instance).

The request object (usually a urllib2.Request instance) must support the methods get_full_url(), get_host(), unverifiable(), and get_origin_req_host(), as documented by urllib2. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set.

CookieJar.set_policy(policy)
Set the CookiePolicy instance to be used.

CookieJar.make_cookies(response, request)
Return sequence of Cookie objects extracted from response object.

See the documentation for extract_cookies() for the interfaces required of the response and request arguments.

CookieJar.set_cookie_if_ok(cookie, request)
Set a Cookie if policy says it's OK to do so.

CookieJar.set_cookie(cookie)
Set a Cookie, without checking with policy to see whether or not it should be set.

CookieJar.clear([domain[, path[, name]]])
Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that domain will be removed. If given two arguments, cookies belonging to the specified domain and

URL path are removed. If given three arguments, then the cookie with the specified domain, path and name is removed.

Raises KeyError if no matching cookie exists.

CookieJar.clear_session_cookies()
Discard all session cookies.

Discards all contained cookies that have a true discard attribute (usually because they had either no max-age or expires cookie-attribute, or an explicit discard cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the save() method won't save session cookies anyway, unless you ask otherwise by passing a true ignore_discard argument.

FileCookieJar implements the following additional methods:

FileCookieJar.save(filename=None, ignore_discard=False, ignore_expires=False)
Save cookies to a file.

This base class raises NotImplementedError. Subclasses may leave this method unimplemented.

filename is the name of file in which to save cookies. If filename is not specified, self.filename is used (whose default is the value passed to the constructor, if any); if self.filename is None, ValueError is raised.

ignore_discard: save even cookies set to be discarded. ignore_expires: save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the load() or revert() methods.

FileCookieJar.load(filename=None, ignore_discard=False, ignore_expires=False)
Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for save().

The named file must be in the format understood by the class, or LoadError will be raised. Also, IOError may be raised, for example if the file does not exist.

Note: For backwards-compatibility with Python 2.4 (which raised an IOError), LoadError is a subclass of IOError.

FileCookieJar.revert(filename=None, ignore_discard=False, ignore_expires=False)
Clear all cookies and reload cookies from a saved file.

revert() can raise the same exceptions as load(). If there is a failure, the object's state will not be altered.

FileCookieJar instances have the following public attributes:

FileCookieJar.filename
Filename of default file in which to keep cookies. This attribute may be assigned to.

FileCookieJar.delayload
If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only affects performance, not behaviour (unless the cookies on disk are changing). A CookieJar object may ignore it. None of the FileCookieJar classes included in the standard library lazily loads cookies.

20.21.2 FileCookieJar subclasses and co-operation with web browsers

The following CookieJar subclasses are provided for reading and writing.

class cookielib.MozillaCookieJar(filename, delayload=None, policy=None)
A FileCookieJar that can load from and save cookies to disk in the Mozilla cookies.txt file format (which is also used by the Lynx and Netscape browsers).

Note: Version 3 of the Firefox web browser no longer writes cookies in the cookies.txt file format.

Note: This loses information about RFC 2965 cookies, and also about newer or non-standard cookie-attributes such as port.

Warning: Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

class cookielib.LWPCookieJar(filename, delayload=None, policy=None)
A FileCookieJar that can load from and save cookies to disk in format compatible with the libwww-perl library's Set-Cookie3 file format. This is convenient if you want to store cookies in a human-readable file.

20.21.3 CookiePolicy Objects

Objects implementing the CookiePolicy interface have the following methods:

CookiePolicy.set_ok(cookie, request)
Return boolean value indicating whether cookie should be accepted from server.

cookie is a cookielib.Cookie instance. request is an object implementing the interface defined by the documentation for CookieJar.extract_cookies().

CookiePolicy.return_ok(cookie, request)
Return boolean value indicating whether cookie should be returned to server.

cookie is a cookielib.Cookie instance. request is an object implementing the interface defined by the documentation for CookieJar.add_cookie_header().

CookiePolicy.domain_return_ok(domain, request)
Return false if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from domain_return_ok() and path_return_ok() leaves all the work to return_ok().

If domain_return_ok() returns true for the cookie domain, path_return_ok() is called for the cookie path. Otherwise, path_return_ok() and return_ok() are never called for that cookie domain. If path_return_ok() returns true, return_ok() is called with the Cookie object itself for a full check. Otherwise, return_ok() is never called for that cookie path.

Note that domain_return_ok() is called for every cookie domain, not just for the request domain. For example, the function might be called with both ".example.com" and "www.example.com" if the request domain is "www.example.com". The same goes for path_return_ok().

The request argument is as documented for return_ok().

CookiePolicy.path_return_ok(path, request)
Return false if cookies should not be returned, given cookie path.

See the documentation for domain_return_ok().

In addition to implementing the methods above, implementations of the CookiePolicy interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

CookiePolicy.netscape
Implement Netscape protocol.

CookiePolicy.rfc2965
Implement RFC 2965 protocol.

CookiePolicy.hide_cookie2
Don't add Cookie2 header to requests (the presence of this header indicates to the server that we understand RFC 2965 cookies).

The most useful way to define a CookiePolicy class is by subclassing from DefaultCookiePolicy and overriding some or all of the methods above. CookiePolicy itself may be used as a 'null policy' to allow setting and receiving any and all cookies (this is unlikely to be useful).

20.21.4 DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both RFC 2965 and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks:

```
import cookielib
class MyCookiePolicy(cookielib.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not cookielib.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the CookiePolicy interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blacklist and whitelist is provided (both off by default). Only domains not in the blacklist and present in the whitelist (if the whitelist is active) participate in cookie setting and returning. Use the blocked_domains constructor argument, and blocked_domains() and set_blocked_domains() methods (and the corresponding argument and methods for allowed_domains). If you set a whitelist, you can turn it off again by setting it to None.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, "example.com" matches a blacklist entry of "example.com", but "www.example.com" does

not. Domains that do start with a dot are matched by more specific domains too. For example, both "www.example.com" and "www.coyote.example.com" match ".example.com" (but "example.com" itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains "192.168.1.2" and ".168.1.2", 192.168.1.2 is blocked, but 193.168.1.2 is not.

DefaultCookiePolicy implements the following additional methods:

DefaultCookiePolicy.blocked_domains()
Return the sequence of blocked domains (as a tuple).

DefaultCookiePolicy.set_blocked_domains(blocked_domains)
Set the sequence of blocked domains.

DefaultCookiePolicy.is_blocked(domain)
Return whether domain is on the blacklist for setting or receiving cookies.

DefaultCookiePolicy.allowed_domains()
Return None, or the sequence of allowed domains (as a tuple).

DefaultCookiePolicy.set_allowed_domains(allowed_domains)
Set the sequence of allowed domains, or None.

DefaultCookiePolicy.is_not_allowed(domain)
Return whether domain is not on the whitelist for setting or receiving cookies.

DefaultCookiePolicy instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

DefaultCookiePolicy.rfc2109_as_netscape
If true, request that the CookieJar instance downgrade RFC 2109 cookies (ie. cookies received in a Set-Cookie header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the Cookie instance to 0. The default value is None, in which case RFC 2109 cookies are downgraded if and only if RFC 2965 handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

New in version 2.5.

General strictness switches:

DefaultCookiePolicy.strict_domain
Don't allow sites to set two-component domains with country-code top-level domains like .co.uk, .gov.uk, .co.nz.etc. This is far from perfect and isn't guaranteed to work!

RFC 2965 protocol strictness switches:

DefaultCookiePolicy.strict_rfc2965_unverifiable
Follow RFC 2965 rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a redirect or a request for an image hosted on another site). If this is false, cookies are never blocked on the basis of verifiability

Netscape protocol strictness switches:

DefaultCookiePolicy.strict_ns_unverifiable
Apply RFC 2965 rules on unverifiable transactions even to Netscape cookies.

DefaultCookiePolicy.strict_ns_domain
Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

DefaultCookiePolicy.strict_ns_set_initial_dollar
Ignore cookies in Set-Cookie: headers that have names starting with '$'.

DefaultCookiePolicy.strict_ns_set_path
: Don't allow setting cookies whose path doesn't path-match request URI.

strict_ns_domain is a collection of flags. Its value is constructed by or-ing together (for example, DomainStrictNoDots|DomainStrictNonDomain means both flags are set).

DefaultCookiePolicy.DomainStrictNoDots
: When setting cookies, the 'host prefix' must not contain a dot (eg. www.foo.bar.com can't set a cookie for .bar.com, because www.foo contains a dot).

DefaultCookiePolicy.DomainStrictNonDomain
: Cookies that did not explicitly specify a domain cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. spam.example.com won't be returned cookies from example.com that had no domain cookie-attribute).

DefaultCookiePolicy.DomainRFC2965Match
: When setting cookies, require a full RFC 2965 domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags:

DefaultCookiePolicy.DomainLiberal
: Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

DefaultCookiePolicy.DomainStrict
: Equivalent to DomainStrictNoDots|DomainStrictNonDomain.

20.21.5 Cookie Objects

Cookie instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the max-age and expires cookie-attributes contain equivalent information, and because RFC 2109 cookies may be 'downgraded' by cookielib from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a CookiePolicy method. The class does not enforce internal consistency, so you should know what you're doing if you do that.

Cookie.version
: Integer or None. Netscape cookies have version 0. RFC 2965 and RFC 2109 cookies have a version cookie-attribute of 1. However, note that cookielib may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case version is 0.

Cookie.name
: Cookie name (a string).

Cookie.value
: Cookie value (a string), or None.

Cookie.port
: String representing a port or a set of ports (eg. '80', or '80,8080'), or None.

Cookie.path
: Cookie path (a string, eg. `'/acme/rocket_launchers'`).

Cookie.secure
: True if cookie should only be returned over a secure connection.

Cookie.expires
: Integer expiry date in seconds since epoch, or None. See also the is_expired() method.

Cookie.discard
: True if this is a session cookie.

Cookie.comment
: String comment from the server explaining the function of this cookie, or None.

Cookie.comment_url
: URL linking to a comment from the server explaining the function of this cookie, or None.

Cookie.rfc2109
: True if this cookie was received as an RFC 2109 cookie (ie. the cookie arrived in a Set-Cookie header, and the value of the Version cookie-attribute in that header was 1). This attribute is provided because cookielib may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case version is 0.

  New in version 2.5.

Cookie.port_specified
: True if a port or set of ports was explicitly specified by the server (in the Set-Cookie / Set-Cookie2 header).

Cookie.domain_specified
: True if a domain was explicitly specified by the server.

Cookie.domain_initial_dot
: True if the domain explicitly specified by the server began with a dot (`'.'`).

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods:

Cookie.has_nonstandard_attr(name)
: Return true if cookie has the named cookie-attribute.

Cookie.get_nonstandard_attr(name, default=None)
: If cookie has the named cookie-attribute, return its value. Otherwise, return default.

Cookie.set_nonstandard_attr(name, value)
: Set the value of the named cookie-attribute.

The Cookie class also defines the following method:

Cookie.is_expired([now=None])
: True if cookie has passed the time at which the server requested it should expire. If now is given (in seconds since the epoch), return whether the cookie has expired at the specified time.

20.21.6 Examples

The first example shows the most common usage of cookielib:

```
import cookielib, urllib2
cj = cookielib.CookieJar()
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file):

```
import os, cookielib, urllib2
cj = cookielib.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
```

(continues on next page)

(continued from previous page)

```
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

The next example illustrates the use of DefaultCookiePolicy. Turn on RFC 2965 cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned:

```
import urllib2
from cookielib import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=DefaultCookiePolicy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

20.22 Cookie — HTTP state management

Note: The Cookie module has been renamed to http.cookies in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/Cookie.py

The Cookie module defines classes for abstracting the concept of cookies, an HTTP state management mechanism. It supports both simple string-only cookies, and provides an abstraction for having any serializable data-type as cookie value.

The module formerly strictly applied the parsing rules described in the RFC 2109 and RFC 2068 specifications. It has since been discovered that MSIE 3.0x doesn't follow the character rules outlined in those specs and also many current day browsers and servers have relaxed parsing rules when comes to Cookie handling. As a result, the parsing rules used are a bit less strict.

The character set, string.ascii_letters, string.digits and !#$%&'*+-.^_`|~ denote the set of valid characters allowed by this module in Cookie name (as key).

Note: On encountering an invalid cookie, CookieError is raised, so if your cookie data comes from a browser you should always prepare for invalid data and catch CookieError on parsing.

---

exception Cookie.CookieError
: Exception failing because of RFC 2109 invalidity: incorrect attributes, incorrect Set-Cookie header, etc.

class Cookie.BaseCookie([input])
: This class is a dictionary-like object whose keys are strings and whose values are Morsel instances. Note that upon setting a key to a value, the value is first converted to a Morsel containing the key and the value.

  If input is given, it is passed to the load() method.

class Cookie.SimpleCookie([input])
This class derives from BaseCookie and overrides value_decode() and value_encode() to be the identity and str() respectively.

class Cookie.SerialCookie([input])
This class derives from BaseCookie and overrides value_decode() and value_encode() to be the pickle.loads() and pickle.dumps().

Deprecated since version 2.3: Reading pickled values from untrusted cookie data is a huge security hole, as pickle strings can be crafted to cause arbitrary code to execute on your server. It is supported for backwards compatibility only, and may eventually go away.

class Cookie.SmartCookie([input])
This class derives from BaseCookie. It overrides value_decode() to be pickle.loads() if it is a valid pickle, and otherwise the value itself. It overrides value_encode() to be pickle.dumps() unless it is a string, in which case it returns the value itself.

Deprecated since version 2.3: The same security warning from SerialCookie applies here.

A further security note is warranted. For backwards compatibility, the Cookie module exports a class named Cookie which is just an alias for SmartCookie. This is probably a mistake and will likely be removed in a future version. You should not use the Cookie class in your applications, for the same reason why you should not use the SerialCookie class.

See also:

Module cookielib HTTP cookie handling for web clients. The cookielib and Cookie modules do not depend on each other.

RFC 2109 - HTTP State Management Mechanism This is the state management specification implemented by this module.

20.22.1 Cookie Objects

BaseCookie.value_decode(val)
Return a decoded value from a string representation. Return value can be any type. This method does nothing in BaseCookie — it exists so it can be overridden.

BaseCookie.value_encode(val)
Return an encoded value. val can be any type, but return value must be a string. This method does nothing in BaseCookie — it exists so it can be overridden.

In general, it should be the case that value_encode() and value_decode() are inverses on the range of value_decode.

BaseCookie.output([attrs[, header[, sep]]])
Return a string representation suitable to be sent as HTTP headers. attrs and header are sent to each Morsel's output() method. sep is used to join the headers together, and is by default the combination '\r\n' (CRLF).

Changed in version 2.5: The default separator has been changed from '\n' to match the cookie specification.

BaseCookie.js_output([attrs])
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP headers was sent.

The meaning for attrs is the same as in output().

BaseCookie.load(rawdata)
If rawdata is a string, parse it as an HTTP_COOKIE and add the values found there as Morsels. If it is a dictionary, it is equivalent to:

```
for k, v in rawdata.items():
    cookie[k] = v
```

20.22.2 Morsel Objects

class Cookie.Morsel
Abstract a key/value pair, which has some RFC 2109 attributes.

Morsels are dictionary-like objects, whose set of keys is constant — the valid RFC 2109 attributes, which are

- expires
- path
- comment
- domain
- max-age
- secure
- version
- httponly

The attribute httponly specifies that the cookie is only transferred in HTTP requests, and is not accessible through JavaScript. This is intended to mitigate some forms of cross-site scripting.

The keys are case-insensitive.

New in version 2.6: The httponly attribute was added.

Morsel.value
The value of the cookie.

Morsel.coded_value
The encoded value of the cookie — this is what should be sent.

Morsel.key
The name of the cookie.

Morsel.set(key, value, coded_value)
Set the key, value and coded_value attributes.

Morsel.isReservedKey(K)
Whether K is a member of the set of keys of a Morsel.

Morsel.output([attrs[, header]])
Return a string representation of the Morsel, suitable to be sent as an HTTP header. By default, all the attributes are included, unless attrs is given, in which case it should be a list of attributes to use. header is by default "Set-Cookie:".

Morsel.js_output([attrs])
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP header was sent.

The meaning for attrs is the same as in output().

Morsel.OutputString([attrs])
: Return a string representing the Morsel, without any surrounding HTTP or JavaScript.

  The meaning for attrs is the same as in output().

20.22.3 Example

The following example demonstrates how to use the Cookie module.

```
>>> import Cookie
>>> C = Cookie.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print C # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print C.output() # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = Cookie.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print C.output(header="Cookie:")
Cookie: rocky=road; Path=/cookie
>>> print C.output(attrs=[], header="Cookie:")
Cookie: rocky=road
>>> C = Cookie.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print C
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = Cookie.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print C
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = Cookie.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print C
Set-Cookie: oreo=doublestuff; Path=/
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = Cookie.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number=7
Set-Cookie: string=seven
>>> # SerialCookie and SmartCookie are deprecated
>>> # using it can cause security loopholes in your code.
>>> C = Cookie.SerialCookie()
>>> C["number"] = 7
```

(continues on next page)

(continued from previous page)

```
>>> C["string"] = "seven"
>>> C["number"].value
7
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number="I7\012."
Set-Cookie: string="S'seven'\012p1\012."
>>> C = Cookie.SmartCookie()
>>> C["number"] = 7
>>> C["string"] = "seven"
>>> C["number"].value
7
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number="I7\012."
Set-Cookie: string=seven
```

20.23 xmlrpclib — XML-RPC client access

Note: The xmlrpclib module has been renamed to xmlrpc.client in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

New in version 2.2.

Source code: Lib/xmlrpclib.py

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

Warning: The xmlrpclib module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

Changed in version 2.7.9: For HTTPS URIs, xmlrpclib now performs all the necessary certificate and hostname checks by default.

class xmlrpclib.ServerProxy(uri[, transport[, encoding[, verbose[, allow_none[, use_datetime[, context]]]]]])

A ServerProxy instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal SafeTransport instance for https: URLs and an internal HTTP Transport instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If allow_none is true, the Python constant None will be translated into XML; the default behaviour is for None to raise a TypeError. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The use_datetime flag can be used to cause date/time values to be presented as datetime.datetime objects; this is false by default. datetime.datetime objects may be passed to calls.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: http://user:pass@host:port/path. The user:pass portion will be base64-encoded as an HTTP 'Authorization' header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, context may be ssl.SSLContext and configures the SSL settings of the underlying HTTPS connection.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

XML-RPC type	Python type
boolean	bool
int or i4	int or long in range from -2147483648 to 2147483647.
double	float
string	str or unicode
array	list or tuple containing conformable elements. Arrays are returned as lists.
struct	dict. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their __dict__ attribute is transmitted.
dateTime.iso8601	DateTime or datetime.datetime. Returned type depends on values of the use_datetime flags.
base64	Binary
nil	The None constant. Passing is allowed only if allow_none is true.

This is the full set of data types supported by XML-RPC. Method calls may also raise a special Fault instance, used to signal XML-RPC server errors, or ProtocolError used to signal an error in the HTTP/HTTPS transport layer. Both Fault and ProtocolError derive from a base class called Error. Note that even though starting with Python 2.2 you can subclass built-in types, the xmlrpclib module currently does not marshal instances of such subclasses.

When passing strings, characters special to XML such as <, >, and & will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary strings via XML-RPC, use the Binary wrapper class described below.

Server is retained as an alias for ServerProxy for backwards compatibility. New code should use ServerProxy.

Changed in version 2.5: The use_datetime flag was added.

Changed in version 2.6: Instances of new-style classes can be passed in if they have an __dict__ attribute and don't have a base class that is marshalled in a special way.

Changed in version 2.7.9: Added the context argument.

See also:

XML-RPC HOWTO A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC Introspection Describes the XML-RPC protocol extension for introspection.

XML-RPC Specification The official specification.

Unofficial XML-RPC Errata Fredrik Lundh's "unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at 'best practices' to use when designing your own XML-RPC implementations."

20.23.1 ServerProxy Objects

A ServerProxy instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a Fault or ProtocolError object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved system attribute:

ServerProxy.system.listMethods()
This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

ServerProxy.system.methodSignature(name)
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply "string, array". If it expects three integers and returns a string, its signature is "string, int, int, int".

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

ServerProxy.system.methodHelp(name)
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

20.23.2 Boolean Objects

This class may be initialized from any Python value; the instance returned depends only on its truth value. It supports various Python operators through __cmp__(), __repr__(), __int__(), and __nonzero__() methods, all implemented in the obvious ways.

It also has the following method, supported mainly for internal use by the unmarshalling code:

Boolean.encode(out)
: Write the XML-RPC encoding of this Boolean item to the out stream object.

A working example follows. The server code:

```
import xmlrpclib
from SimpleXMLRPCServer import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
print "3 is even: %s" % str(proxy.is_even(3))
print "100 is even: %s" % str(proxy.is_even(100))
```

20.23.3 DateTime Objects

class xmlrpclib.DateTime
: This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a datetime.datetime instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

  decode(string)
  : Accept a string as the instance's new time value.

  encode(out)
  : Write the XML-RPC encoding of this DateTime item to the out stream object.

  It also supports certain of Python's built-in operators through __cmp__() and __repr__() methods.

A working example follows. The server code:

```
import datetime
from SimpleXMLRPCServer import SimpleXMLRPCServer
import xmlrpclib

def today():
    today = datetime.datetime.today()
    return xmlrpclib.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(today, "today")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpclib
import datetime

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print "Today: %s" % converted.strftime("%d.%m.%Y, %H:%M")
```

20.23.4 Binary Objects

class xmlrpclib.Binary

This class may be initialized from string data (which may include NULs). The primary access to the content of a Binary object is provided by an attribute:

data

The binary data encapsulated by the Binary instance. The data is provided as an 8-bit string.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(string)

Accept a base64 string and decode it as the instance's new data.

encode(out)

Write the XML-RPC base 64 encoding of this binary item to the out stream object.

The encoded data will have newlines every 76 characters as per RFC 2045 section 6.8, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python's built-in operators through a __cmp__() method.

Example usage of the binary objects. We're going to transfer an image over XMLRPC:

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
import xmlrpclib

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpclib.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

The client gets the image and saves it to a file:

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

20.23.5 Fault Objects

class xmlrpclib.Fault
A Fault object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

faultCode
A string indicating the fault type.

faultString
A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a Fault by returning a complex type object. The server code:

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(add, 'add')

server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpclib.Fault as err:
    print "A fault occurred"
    print "Fault code: %d" % err.faultCode
    print "Fault string: %s" % err.faultString
```

20.23.6 ProtocolError Objects

class xmlrpclib.ProtocolError
A ProtocolError object describes a protocol error in the underlying transport layer (such as a 404 'not found' error if the server named by the URI does not exist). It has the following attributes:

url
The URI or URL that triggered the error.

errcode
The error code.

errmsg
The error message or diagnostic string.

headers
A string containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we're going to intentionally cause a ProtocolError by providing a URI that doesn't point to an XMLRPC server:

```
import xmlrpclib

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpclib.ServerProxy("http://www.google.com/")

try:
    proxy.some_method()
except xmlrpclib.ProtocolError as err:
    print "A protocol error occurred"
    print "URL: %s" % err.url
    print "HTTP/HTTPS headers: %s" % err.headers
    print "Error code: %d" % err.errcode
    print "Error message: %s" % err.errmsg
```

20.23.7 MultiCall Objects

New in version 2.4.

The MultiCall object provides a way to encapsulate multiple calls to a remote server into a single request[1].

class xmlrpclib.MultiCall(server)

Create an object used to boxcar method calls. server is the eventual target of the call. Calls can be made to the result object, but they will immediately return None, and only store the call name and parameters in the MultiCall object. Calling the object itself causes all stored calls to be transmitted as a single system.multicall request. The result of this call is a generator; iterating over this generator yields the individual results.

A usage example of this class follows. The server code

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

def add(x,y):
    return x+y

def subtract(x, y):
    return x-y

def multiply(x, y):
    return x*y

def divide(x, y):
    return x/y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

[1] This approach has been first presented in a discussion on xmlrpc.com.

The client code for the preceding server:

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
multicall = xmlrpclib.MultiCall(proxy)
multicall.add(7,3)
multicall.subtract(7,3)
multicall.multiply(7,3)
multicall.divide(7,3)
result = multicall()

print "7+3=%d, 7-3=%d, 7*3=%d, 7/3=%d" % tuple(result)
```

20.23.8 Convenience Functions

xmlrpclib.boolean(value)
: Convert any Python value to one of the XML-RPC Boolean constants, True or False.

xmlrpclib.dumps(params[, methodname[, methodresponse[, encoding[, allow_none]]]])
: Convert params into an XML-RPC request. or into a response if methodresponse is true. params can be either a tuple of arguments or an instance of the Fault exception class. If methodresponse is true, only a single value can be returned, meaning that params must be of length 1. encoding, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's None value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for allow_none.

xmlrpclib.loads(data[, use_datetime])
: Convert an XML-RPC request or response into Python objects, a (params, methodname). params is a tuple of argument; methodname is a string, or None if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a Fault exception. The use_datetime flag can be used to cause date/time values to be presented as datetime.datetime objects; this is false by default.

  Changed in version 2.5: The use_datetime flag was added.

20.23.9 Example of Client Usage

```
# simple test program (from the XML-RPC specification)
from xmlrpclib import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
server = ServerProxy("http://betty.userland.com")

print server

try:
    print server.examples.getStateName(41)
except Error as v:
    print "ERROR", v
```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how:

```
import xmlrpclib, httplib

class ProxiedTransport(xmlrpclib.Transport):
    def set_proxy(self, proxy):
        self.proxy = proxy

    def make_connection(self, host):
        self.realhost = host
        h = httplib.HTTPConnection(self.proxy)
        return h

    def send_request(self, connection, handler, request_body):
        connection.putrequest("POST", 'http://%s%s' % (self.realhost, handler))

    def send_host(self, connection, host):
        connection.putheader('Host', self.realhost)

p = ProxiedTransport()
p.set_proxy('proxy-server:8080')
server = xmlrpclib.ServerProxy('http://time.xmlrpc.com/RPC2', transport=p)
print server.currentTime.getCurrentTime()
```

20.23.10 Example of Client and Server Usage

See SimpleXMLRPCServer Example.

20.24 SimpleXMLRPCServer — Basic XML-RPC server

Note: The SimpleXMLRPCServer module has been merged into xmlrpc.server in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

New in version 2.2.

Source code: Lib/SimpleXMLRPCServer.py

The SimpleXMLRPCServer module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using SimpleXMLRPCServer, or embedded in a CGI environment, using CGIXMLRPCRequestHandler.

class SimpleXMLRPCServer.SimpleXMLRPCServer(addr[, requestHandler[, logRequests[, allow_none[, encoding[, bind_and_activate]]]]])

Create a new server instance. This class provides methods for registration of functions that can be called by the XML-RPC protocol. The requestHandler parameter should be a factory for request handler instances; it defaults to SimpleXMLRPCRequestHandler. The addr and requestHandler parameters are passed to the SocketServer.TCPServer constructor. If logRequests is true (the default), requests will be logged; setting this parameter to false will turn off logging. The allow_none and encoding parameters are passed on to xmlrpclib and control the XML-RPC responses that will be returned from the server. The bind_and_activate parameter controls whether server_bind() and server_activate() are called immediately by the constructor; it defaults to true. Setting it to false allows code to manipulate the allow_reuse_address class variable before the address is bound.

Changed in version 2.5: The allow_none and encoding parameters were added.

Changed in version 2.6: The bind_and_activate parameter was added.

class SimpleXMLRPCServer.CGIXMLRPCRequestHandler([allow_none[, encoding]])
Create a new instance to handle XML-RPC requests in a CGI environment. The allow_none and encoding parameters are passed on to xmlrpclib and control the XML-RPC responses that will be returned from the server.

New in version 2.3.

Changed in version 2.5: The allow_none and encoding parameters were added.

class SimpleXMLRPCServer.SimpleXMLRPCRequestHandler
Create a new request handler instance. This request handler supports POST requests and modifies logging so that the logRequests parameter to the SimpleXMLRPCServer constructor parameter is honored.

20.24.1 SimpleXMLRPCServer Objects

The SimpleXMLRPCServer class is based on SocketServer.TCPServer and provides a means of creating simple, stand alone XML-RPC servers.

SimpleXMLRPCServer.register_function(function[, name])
Register a function that can respond to XML-RPC requests. If name is given, it will be the method name associated with function, otherwise function.__name__ will be used. name can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

SimpleXMLRPCServer.register_instance(instance[, allow_dotted_names])
Register an object which is used to expose method names which have not been registered using register_function(). If instance contains a _dispatch() method, it is called with the requested method name and the parameters from the request. Its API is def _dispatch(self, method, params) (note that params does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as func(*params), expanding the parameter list. The return value from _dispatch() is returned to the client as the result. If instance does not have a _dispatch() method, it is searched for an attribute matching the name of the requested method.

If the optional allow_dotted_names argument is true and the instance does not have a _dispatch() method, then if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

> Warning: Enabling the allow_dotted_names option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this option on a secure, closed network.

Changed in version 2.3.5,: 2.4.1 allow_dotted_names was added to plug a security hole; prior versions are insecure.

SimpleXMLRPCServer.register_introspection_functions()
Registers the XML-RPC introspection functions system.listMethods, system.methodHelp and system.methodSignature.

New in version 2.3.

SimpleXMLRPCServer.register_multicall_functions()
: Registers the XML-RPC multicall function system.multicall.

SimpleXMLRPCRequestHandler.rpc_paths
: An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 "no such page" HTTP error. If this tuple is empty, all paths will be considered valid. The default value is `('/', '/RPC2')`.

  New in version 2.5.

SimpleXMLRPCRequestHandler.encode_threshold
: If this attribute is not None, responses larger than this value will be encoded using the gzip transfer encoding, if permitted by the client. The default is 1400 which corresponds roughly to a single TCP packet.

  New in version 2.7.

SimpleXMLRPCServer Example

Server code:

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
from SimpleXMLRPCServer import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
server = SimpleXMLRPCServer(("localhost", 8000),
                            requestHandler=RequestHandler)
server.register_introspection_functions()

# Register pow() function; this will use the value of
# pow.__name__ as the name, which is just 'pow'.
server.register_function(pow)

# Register a function under a different name
def adder_function(x,y):
    return x + y
server.register_function(adder_function, 'add')

# Register an instance; all the methods of the instance are
# published as XML-RPC methods (in this case, just 'div').
class MyFuncs:
    def div(self, x, y):
        return x // y

server.register_instance(MyFuncs())

# Run the server's main loop
server.serve_forever()
```

The following client code will call the methods made available by the preceding server:

```
import xmlrpclib

s = xmlrpclib.ServerProxy('http://localhost:8000')
```

(continues on next page)

(continued from previous page)

```
print s.pow(2,3)  # Returns 2**3 = 8
print s.add(2,3)  # Returns 5
print s.div(5,2)  # Returns 5//2 = 2

# Print list of available methods
print s.system.listMethods()
```

The following SimpleXMLRPCServer example is included in the module Lib/SimpleXMLRPCServer.py:

```
server = SimpleXMLRPCServer(("localhost", 8000))
server.register_function(pow)
server.register_function(lambda x,y: x+y, 'add')
server.register_multicall_functions()
server.serve_forever()
```

This demo server can be run from the command line as:

```
python -m SimpleXMLRPCServer
```

Example client code which talks to the above server is included with Lib/xmlrpclib.py:

```
server = ServerProxy("http://localhost:8000")
print server
multi = MultiCall(server)
multi.pow(2, 9)
multi.add(5, 1)
multi.add(24, 11)
try:
    for response in multi():
        print response
except Error, v:
    print "ERROR", v
```

And the client can be invoked directly using the following command:

```
python -m xmlrpclib
```

20.24.2 CGIXMLRPCRequestHandler

The CGIXMLRPCRequestHandler class can be used to handle XML-RPC requests sent to Python CGI scripts.

CGIXMLRPCRequestHandler.register_function(function[, name])
: Register a function that can respond to XML-RPC requests. If name is given, it will be the method name associated with function, otherwise function.__name__ will be used. name can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

CGIXMLRPCRequestHandler.register_instance(instance)
: Register an object which is used to expose method names which have not been registered using register_function(). If instance contains a _dispatch() method, it is called with the requested method name and the parameters from the request; the return value is returned to the client as the result. If instance does not have a _dispatch() method, it is searched for an attribute matching the name of the requested method; if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The

value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

CGIXMLRPCRequestHandler.register_introspection_functions()
Register the XML-RPC introspection functions system.listMethods, system.methodHelp and system.methodSignature.

CGIXMLRPCRequestHandler.register_multicall_functions()
Register the XML-RPC multicall function system.multicall.

CGIXMLRPCRequestHandler.handle_request([request_text = None])
Handle an XML-RPC request. If request_text is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.

Example:

```
class MyFuncs:
    def div(self, x, y): return x // y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

20.25 DocXMLRPCServer — Self-documenting XML-RPC server

Note: The DocXMLRPCServer module has been merged into xmlrpc.server in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

New in version 2.3.

The DocXMLRPCServer module extends the classes found in SimpleXMLRPCServer to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using DocXMLRPCServer, or embedded in a CGI environment, using DocCGIXMLRPCRequestHandler.

class DocXMLRPCServer.DocXMLRPCServer(addr[, requestHandler[, logRequests[, allow_none[, encoding[, bind_and_activate]]]]])
Create a new server instance. All parameters have the same meaning as for SimpleXMLRPCServer.SimpleXMLRPCServer; requestHandler defaults to DocXMLRPCRequestHandler.

class DocXMLRPCServer.DocCGIXMLRPCRequestHandler
Create a new instance to handle XML-RPC requests in a CGI environment.

class DocXMLRPCServer.DocXMLRPCRequestHandler
Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the logRequests parameter to the DocXMLRPCServer constructor parameter is honored.

20.25.1 DocXMLRPCServer Objects

The DocXMLRPCServer class is derived from SimpleXMLRPCServer.SimpleXMLRPCServer and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocXMLRPCServer.set_server_title(server_title)
: Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocXMLRPCServer.set_server_name(server_name)
: Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocXMLRPCServer.set_server_documentation(server_documentation)
: Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

20.25.2 DocCGIXMLRPCRequestHandler

The DocCGIXMLRPCRequestHandler class is derived from SimpleXMLRPCServer.CGIXMLRPCRequestHandler and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocCGIXMLRPCRequestHandler.set_server_title(server_title)
: Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocCGIXMLRPCRequestHandler.set_server_name(server_name)
: Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocCGIXMLRPCRequestHandler.set_server_documentation(server_documentation)
: Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

CHAPTER

TWENTYONE

MULTIMEDIA SERVICES

The modules described in this chapter implement various algorithms or interfaces that are mainly useful for multimedia applications. They are available at the discretion of the installation. Here's an overview:

21.1 audioop — Manipulate raw audio data

The audioop module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16 or 32 bits wide, stored in Python strings. This is the same format as used by the al and sunaudiodev modules. All scalar items are integers, unless specified otherwise.

This module provides support for a-LAW, u-LAW and Intel/DVI ADPCM encodings.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

exception audioop.error
: This exception is raised on all errors, such as unknown number of bytes per sample, etc.

audioop.add(fragment1, fragment2, width)
: Return a fragment which is the addition of the two samples passed as parameters. width is the sample width in bytes, either 1, 2 or 4. Both fragments should have the same length. Samples are truncated in case of overflow.

audioop.adpcm2lin(adpcmfragment, width, state)
: Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of lin2adpcm() for details on ADPCM coding. Return a tuple (sample, newstate) where the sample has the width specified in width.

audioop.alaw2lin(fragment, width)
: Convert sound fragments in a-LAW encoding to linearly encoded sound fragments. a-LAW encoding always uses 8 bits samples, so width refers only to the sample width of the output fragment here.

  New in version 2.5.

audioop.avg(fragment, width)
: Return the average over all samples in the fragment.

audioop.avgpp(fragment, width)
: Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

audioop.bias(fragment, width, bias)
: Return a fragment that is the original fragment with a bias added to each sample. Samples wrap around in case of overflow.

audioop.cross(fragment, width)
Return the number of zero crossings in the fragment passed as an argument.

audioop.findfactor(fragment, reference)
Return a factor F such that rms(add(fragment, mul(reference, -F))) is minimal, i.e., return the factor with which you should multiply reference to make it match as well as possible to fragment. The fragments should both contain 2-byte samples.

The time taken by this routine is proportional to len(fragment).

audioop.findfit(fragment, reference)
Try to match reference as well as possible to a portion of fragment (which should be the longer fragment). This is (conceptually) done by taking slices out of fragment, using findfactor() to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple (offset, factor) where offset is the (integer) offset into fragment where the optimal match started and factor is the (floating-point) factor as per findfactor().

audioop.findmax(fragment, length)
Search fragment for a slice of length length samples (not bytes!) with maximum energy, i.e., return i for which rms(fragment[i*2:(i+length)*2]) is maximal. The fragments should both contain 2-byte samples.

The routine takes time proportional to len(fragment).

audioop.getsample(fragment, width, index)
Return the value of sample index from the fragment.

audioop.lin2adpcm(fragment, width, state)
Convert samples to 4 bit Intel/DVI ADPCM encoding. ADPCM coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) step. The Intel/DVI ADPCM algorithm has been selected for use by the IMA, so it may well become a standard.

state is a tuple containing the state of the coder. The coder returns a tuple (adpcmfrag, newstate), and the newstate should be passed to the next call of lin2adpcm(). In the initial call, None can be passed as the state. adpcmfrag is the ADPCM coded fragment packed 2 4-bit values per byte.

audioop.lin2alaw(fragment, width)
Convert samples in the audio fragment to a-LAW encoding and return this as a Python string. a-LAW is an audio encoding format whereby you get a dynamic range of about 13 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

New in version 2.5.

audioop.lin2lin(fragment, width, newwidth)
Convert samples between 1-, 2- and 4-byte formats.

Note: In some audio formats, such as .WAV files, 16 and 32 bit samples are signed, but 8 bit samples are unsigned. So when converting to 8 bit wide samples for these formats, you need to also add 128 to the result:

```
new_frames = audioop.lin2lin(frames, old_width, 1)
new_frames = audioop.bias(new_frames, 1, 128)
```

The same, in reverse, has to be applied when converting from 8 to 16 or 32 bit width samples.

---

audioop.lin2ulaw(fragment, width)
Convert samples in the audio fragment to u-LAW encoding and return this as a Python string. u-LAW

is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

audioop.max(fragment, width)
Return the maximum of the absolute value of all samples in a fragment.

audioop.maxpp(fragment, width)
Return the maximum peak-peak value in the sound fragment.

audioop.minmax(fragment, width)
Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

audioop.mul(fragment, width, factor)
Return a fragment that has all samples in the original fragment multiplied by the floating-point value factor. Samples are truncated in case of overflow.

audioop.ratecv(fragment, width, nchannels, inrate, outrate, state[, weightA[, weightB]])
Convert the frame rate of the input fragment.

state is a tuple containing the state of the converter. The converter returns a tuple (newfragment, newstate), and newstate should be passed to the next call of ratecv(). The initial call should pass None as the state.

The weightA and weightB arguments are parameters for a simple digital filter and default to 1 and 0 respectively.

audioop.reverse(fragment, width)
Reverse the samples in a fragment and returns the modified fragment.

audioop.rms(fragment, width)
Return the root-mean-square of the fragment, i.e. sqrt(sum(S_i^2)/n).

This is a measure of the power in an audio signal.

audioop.tomono(fragment, width, lfactor, rfactor)
Convert a stereo fragment to a mono fragment. The left channel is multiplied by lfactor and the right channel by rfactor before adding the two channels to give a mono signal.

audioop.tostereo(fragment, width, lfactor, rfactor)
Generate a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are computed from the mono sample, whereby left channel samples are multiplied by lfactor and right channel samples by rfactor.

audioop.ulaw2lin(fragment, width)
Convert sound fragments in u-LAW encoding to linearly encoded sound fragments. u-LAW encoding always uses 8 bits samples, so width refers only to the sample width of the output fragment here.

Note that operations such as mul() or max() make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that:

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you

should send the initial state (the one you passed to lin2adpcm()) along to the decoder, not the final state (as returned by the coder). If you want to use struct.Struct to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.

The find*() routines might look a bit funny at first sight. They are primarily meant to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample:

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)    # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    #              out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

21.2 imageop — Manipulate raw image data

Deprecated since version 2.6: The imageop module has been removed in Python 3.

The imageop module contains some useful operations on images. It operates on images consisting of 8 or 32 bit pixels stored in Python strings. This is the same format as used by gl.lrectwrite() and the imgfile module.

The module defines the following variables and functions:

exception imageop.error
: This exception is raised on all errors, such as unknown number of bits per pixel, etc.

imageop.crop(image, psize, width, height, x0, y0, x1, y1)
: Return the selected part of image, which should be width by height in size and consist of pixels of psize bytes. x0, y0, x1 and y1 are like the gl.lrectread() parameters, i.e. the boundary is included in the new image. The new boundaries need not be inside the picture. Pixels that fall outside the old image will have their value set to zero. If x0 is bigger than x1 the new image is mirrored. The same holds for the y coordinates.

imageop.scale(image, psize, width, height, newwidth, newheight)
: Return image scaled to size newwidth by newheight. No interpolation is done, scaling is done by simple-minded pixel duplication or removal. Therefore, computer-generated images or dithered images will not look nice after scaling.

imageop.tovideo(image, psize, width, height)
: Run a vertical low-pass filter over an image. It does so by computing each destination pixel as the average of two vertically-aligned source pixels. The main use of this routine is to forestall excessive flicker if the image is displayed on a video device that uses interlacing, hence the name.

imageop.grey2mono(image, width, height, threshold)
: Convert an 8-bit deep greyscale image to a 1-bit deep image by thresholding all the pixels. The resulting image is tightly packed and is probably only useful as an argument to mono2grey().

imageop.dither2mono(image, width, height)
: Convert an 8-bit greyscale image to a 1-bit monochrome image using a (simple-minded) dithering algorithm.

imageop.mono2grey(image, width, height, p0, p1)
: Convert a 1-bit monochrome image to an 8 bit greyscale or color image. All pixels that are zero-valued on input get value p0 on output and all one-value input pixels get value p1 on output. To convert a monochrome black-and-white image to greyscale pass the values 0 and 255 respectively.

imageop.grey2grey4(image, width, height)
: Convert an 8-bit greyscale image to a 4-bit greyscale image without dithering.

imageop.grey2grey2(image, width, height)
: Convert an 8-bit greyscale image to a 2-bit greyscale image without dithering.

imageop.dither2grey2(image, width, height)
: Convert an 8-bit greyscale image to a 2-bit greyscale image with dithering. As for dither2mono(), the dithering algorithm is currently very simple.

imageop.grey42grey(image, width, height)
: Convert a 4-bit greyscale image to an 8-bit greyscale image.

imageop.grey22grey(image, width, height)
: Convert a 2-bit greyscale image to an 8-bit greyscale image.

imageop.backward_compatible
: If set to 0, the functions in this module use a non-backward compatible way of representing multi-byte pixels on little-endian systems. The SGI for which this module was originally written is a big-endian system, so setting this variable will have no effect. However, the code wasn't originally intended to run on anything else, so it made assumptions about byte order which are not universal. Setting this variable to 0 will cause the byte order to be reversed on little-endian systems, so that it then is the same as on big-endian systems.

21.3 aifc — Read and write AIFF and AIFC files

Source code: Lib/aifc.py

This module provides support for reading and writing AIFF and AIFF-C files. AIFF is Audio Interchange File Format, a format for storing digital audio samples in a file. AIFF-C is a newer version of the format that includes the ability to compress the audio data.

Note: Some operations may only work under IRIX; these will raise ImportError when attempting to import the cl module, which is only available on IRIX.

Audio files have a number of parameters that describe the audio data. The sampling rate or frame rate is the number of times per second the sound is sampled. The number of channels indicate if the audio is mono, stereo, or quadro. Each frame consists of one sample per channel. The sample size is the size in bytes of each sample. Thus a frame consists of nchannels*samplesize bytes, and a second's worth of audio consists of nchannels*samplesize*framerate bytes.

For example, CD quality audio has a sample size of two bytes (16 bits), uses two channels (stereo) and has a frame rate of 44,100 frames/second. This gives a frame size of 4 bytes (2*2), and a second's worth occupies 2*2*44100 bytes (176,400 bytes).

Module aifc defines the following function:

aifc.open(file[, mode])
: Open an AIFF or AIFF-C file and return an object instance with methods that are described below. The argument file is either a string naming a file or a file object. mode must be `'r'` or `'rb'` when the file must be opened for reading, or `'w'` or `'wb'` when the file must be opened for writing. If omitted, file.mode is used if it exists, otherwise `'rb'` is used. When used for writing, the file object should be seekable, unless you know ahead of time how many samples you are going to write in total and use writeframesraw() and setnframes().

Objects returned by open() when a file is opened for reading have the following methods:

aifc.getnchannels()
: Return the number of audio channels (1 for mono, 2 for stereo).

aifc.getsampwidth()
: Return the size in bytes of individual samples.

aifc.getframerate()
: Return the sampling rate (number of audio frames per second).

aifc.getnframes()
: Return the number of audio frames in the file.

aifc.getcomptype()
: Return a four-character string describing the type of compression used in the audio file. For AIFF files, the returned value is `'NONE'`.

aifc.getcompname()
: Return a human-readable description of the type of compression used in the audio file. For AIFF files, the returned value is `'not compressed'`.

aifc.getparams()
: Return a tuple consisting of all of the above values in the above order.

aifc.getmarkers()
: Return a list of markers in the audio file. A marker consists of a tuple of three elements. The first is the mark ID (an integer), the second is the mark position in frames from the beginning of the data (an integer), the third is the name of the mark (a string).

aifc.getmark(id)
: Return the tuple as described in getmarkers() for the mark with the given id.

aifc.readframes(nframes)
: Read and return the next nframes frames from the audio file. The returned data is a string containing for each frame the uncompressed samples of all channels.

aifc.rewind()
: Rewind the read pointer. The next readframes() will start from the beginning.

aifc.setpos(pos)
: Seek to the specified frame number.

aifc.tell()
: Return the current frame number.

aifc.close()
: Close the AIFF file. After calling this method, the object can no longer be used.

Objects returned by open() when a file is opened for writing have all the above methods, except for readframes() and setpos(). In addition the following methods exist. The get*() methods can only be called after the corresponding set*() methods have been called. Before the first writeframes() or writeframesraw(), all parameters except for the number of frames must be filled in.

aifc.aiff()
: Create an AIFF file. The default is that an AIFF-C file is created, unless the name of the file ends in `'.aiff'` in which case the default is an AIFF file.

aifc.aifc()
: Create an AIFF-C file. The default is that an AIFF-C file is created, unless the name of the file ends in `'.aiff'` in which case the default is an AIFF file.

aifc.setnchannels(nchannels)
: Specify the number of channels in the audio file.

aifc.setsampwidth(width)
: Specify the size in bytes of audio samples.

aifc.setframerate(rate)
: Specify the sampling frequency in frames per second.

aifc.setnframes(nframes)
: Specify the number of frames that are to be written to the audio file. If this parameter is not set, or not set correctly, the file needs to support seeking.

aifc.setcomptype(type, name)
: Specify the compression type. If not specified, the audio data will not be compressed. In AIFF files, compression is not possible. The name parameter should be a human-readable description of the compression type, the type parameter should be a four-character string. Currently the following compression types are supported: NONE, ULAW, ALAW, G722.

aifc.setparams(nchannels, sampwidth, framerate, comptype, compname)
: Set all the above parameters at once. The argument is a tuple consisting of the various parameters. This means that it is possible to use the result of a getparams() call as argument to setparams().

aifc.setmark(id, pos, name)
: Add a mark with the given id (larger than 0), and the given name at the given position. This method can be called at any time before close().

aifc.tell()
: Return the current write position in the output file. Useful in combination with setmark().

aifc.writeframes(data)
: Write data to the output file. This method can only be called after the audio file parameters have been set.

aifc.writeframesraw(data)
: Like writeframes(), except that the header of the audio file is not updated.

aifc.close()
: Close the AIFF file. The header of the file is updated to reflect the actual size of the audio data. After calling this method, the object can no longer be used.

21.4 sunau — Read and write Sun AU files

Source code: Lib/sunau.py

The sunau module provides a convenient interface to the Sun AU sound format. Note that this module is interface-compatible with the modules aifc and wave.

An audio file consists of a header followed by the data. The fields of the header are:

Field	Contents
magic word	The four bytes .snd.
header size	Size of the header, including info, in bytes.
data size	Physical size of the data, in bytes.
encoding	Indicates how the audio samples are encoded.
sample rate	The sampling rate.
# of channels	The number of channels in the samples.
info	ASCII string giving a description of the audio file (padded with null bytes).

Apart from the info field, all header fields are 4 bytes in size. They are all 32-bit unsigned integers encoded in big-endian byte order.

The sunau module defines the following functions:

sunau.open(file, mode)
: If file is a string, open the file by that name, otherwise treat it as a seekable file-like object. mode can be any of

  `'r'` Read only mode.

  `'w'` Write only mode.

  Note that it does not allow read/write files.

  A mode of `'r'` returns an AU_read object, while a mode of `'w'` or `'wb'` returns an AU_write object.

sunau.openfp(file, mode)
: A synonym for open(), maintained for backwards compatibility.

The sunau module defines the following exception:

exception sunau.Error
: An error raised when something is impossible because of Sun AU specs or implementation deficiency.

The sunau module defines the following data items:

sunau.AUDIO_FILE_MAGIC
: An integer every valid Sun AU file begins with, stored in big-endian form. This is the string .snd interpreted as an integer.

sunau.AUDIO_FILE_ENCODING_MULAW_8
sunau.AUDIO_FILE_ENCODING_LINEAR_8
sunau.AUDIO_FILE_ENCODING_LINEAR_16
sunau.AUDIO_FILE_ENCODING_LINEAR_24
sunau.AUDIO_FILE_ENCODING_LINEAR_32
sunau.AUDIO_FILE_ENCODING_ALAW_8
: Values of the encoding field from the AU header which are supported by this module.

sunau.AUDIO_FILE_ENCODING_FLOAT
sunau.AUDIO_FILE_ENCODING_DOUBLE
sunau.AUDIO_FILE_ENCODING_ADPCM_G721
sunau.AUDIO_FILE_ENCODING_ADPCM_G722
sunau.AUDIO_FILE_ENCODING_ADPCM_G723_3
sunau.AUDIO_FILE_ENCODING_ADPCM_G723_5
: Additional known values of the encoding field from the AU header, but which are not supported by this module.

21.4.1 AU_read Objects

AU_read objects, as returned by open() above, have the following methods:

AU_read.close()
Close the stream, and make the instance unusable. (This is called automatically on deletion.)

AU_read.getnchannels()
Returns number of audio channels (1 for mono, 2 for stereo).

AU_read.getsampwidth()
Returns sample width in bytes.

AU_read.getframerate()
Returns sampling frequency.

AU_read.getnframes()
Returns number of audio frames.

AU_read.getcomptype()
Returns compression type. Supported compression types are `'ULAW'`, `'ALAW'` and `'NONE'`.

AU_read.getcompname()
Human-readable version of getcomptype(). The supported types have the respective names `'CCITT G.711 u-law'`, `'CCITT G.711 A-law'` and `'not compressed'`.

AU_read.getparams()
Returns a tuple (nchannels, sampwidth, framerate, nframes, comptype, compname), equivalent to output of the get*() methods.

AU_read.readframes(n)
Reads and returns at most n frames of audio, as a string of bytes. The data will be returned in linear format. If the original data is in u-LAW format, it will be converted.

AU_read.rewind()
Rewind the file pointer to the beginning of the audio stream.

The following two methods define a term "position" which is compatible between them, and is otherwise implementation dependent.

AU_read.setpos(pos)
Set the file pointer to the specified position. Only values returned from tell() should be used for pos.

AU_read.tell()
Return current file pointer position. Note that the returned value has nothing to do with the actual position in the file.

The following two functions are defined for compatibility with the aifc, and don't do anything interesting.

AU_read.getmarkers()
Returns None.

AU_read.getmark(id)
Raise an error.

21.4.2 AU_write Objects

AU_write objects, as returned by open() above, have the following methods:

AU_write.setnchannels(n)
Set the number of channels.

AU_write.setsampwidth(n)
Set the sample width (in bytes.)

AU_write.setframerate(n)
Set the frame rate.

AU_write.setnframes(n)
Set the number of frames. This can be later changed, when and if more frames are written.

AU_write.setcomptype(type, name)
Set the compression type and description. Only `'NONE'` and `'ULAW'` are supported on output.

AU_write.setparams(tuple)
The tuple should be (nchannels, sampwidth, framerate, nframes, comptype, compname), with values valid for the set*() methods. Set all parameters.

AU_write.tell()
Return current position in the file, with the same disclaimer for the AU_read.tell() and AU_read.setpos() methods.

AU_write.writeframesraw(data)
Write audio frames, without correcting nframes.

AU_write.writeframes(data)
Write audio frames and make sure nframes is correct.

AU_write.close()
Make sure nframes is correct, and close the file.

This method is called upon deletion.

Note that it is invalid to set any parameters after calling writeframes() or writeframesraw().

21.5 wave — Read and write WAV files

Source code: Lib/wave.py

The wave module provides a convenient interface to the WAV sound format. It does not support compression/decompression, but it does support mono/stereo.

The wave module defines the following function and exception:

wave.open(file[, mode])
If file is a string, open the file by that name, otherwise treat it as a seekable file-like object. mode can be any of

`'r'`, `'rb'` Read only mode.

`'w'`, `'wb'` Write only mode.

Note that it does not allow read/write WAV files.

A mode of `'r'` or `'rb'` returns a Wave_read object, while a mode of `'w'` or `'wb'` returns a Wave_write object. If mode is omitted and a file-like object is passed as file, file.mode is used as the default value for mode (the `'b'` flag is still added if necessary).

If you pass in a file-like object, the wave object will not close it when its close() method is called; it is the caller's responsibility to close the file object.

wave.openfp(file, mode)
A synonym for open(), maintained for backwards compatibility.

exception wave.Error
An error raised when something is impossible because it violates the WAV specification or hits an implementation deficiency.

21.5.1 Wave_read Objects

Wave_read objects, as returned by open(), have the following methods:

Wave_read.close()
Close the stream if it was opened by wave, and make the instance unusable. This is called automatically on object collection.

Wave_read.getnchannels()
Returns number of audio channels (1 for mono, 2 for stereo).

Wave_read.getsampwidth()
Returns sample width in bytes.

Wave_read.getframerate()
Returns sampling frequency.

Wave_read.getnframes()
Returns number of audio frames.

Wave_read.getcomptype()
Returns compression type (`'NONE'` is the only supported type).

Wave_read.getcompname()
Human-readable version of getcomptype(). Usually `'not compressed'` parallels `'NONE'`.

Wave_read.getparams()
Returns a tuple (nchannels, sampwidth, framerate, nframes, comptype, compname), equivalent to output of the get*() methods.

Wave_read.readframes(n)
Reads and returns at most n frames of audio, as a string of bytes.

Wave_read.rewind()
Rewind the file pointer to the beginning of the audio stream.

The following two methods are defined for compatibility with the aifc module, and don't do anything interesting.

Wave_read.getmarkers()
Returns None.

Wave_read.getmark(id)
Raise an error.

The following two methods define a term "position" which is compatible between them, and is otherwise implementation dependent.

Wave_read.setpos(pos)
Set the file pointer to the specified position.

Wave_read.tell()
Return current file pointer position.

21.5.2 Wave_write Objects

Wave_write objects, as returned by open(), have the following methods:

Wave_write.close()
Make sure nframes is correct, and close the file if it was opened by wave. This method is called upon object collection.

Wave_write.setnchannels(n)
Set the number of channels.

Wave_write.setsampwidth(n)
Set the sample width to n bytes.

Wave_write.setframerate(n)
Set the frame rate to n.

Wave_write.setnframes(n)
Set the number of frames to n. This will be changed later if more frames are written.

Wave_write.setcomptype(type, name)
Set the compression type and description. At the moment, only compression type NONE is supported, meaning no compression.

Wave_write.setparams(tuple)
The tuple should be (nchannels, sampwidth, framerate, nframes, comptype, compname), with values valid for the set*() methods. Sets all parameters.

Wave_write.tell()
Return current position in the file, with the same disclaimer for the Wave_read.tell() and Wave_read.setpos() methods.

Wave_write.writeframesraw(data)
Write audio frames, without correcting nframes.

Wave_write.writeframes(data)
Write audio frames and make sure nframes is correct.

Note that it is invalid to set any parameters after calling writeframes() or writeframesraw(), and any attempt to do so will raise wave.Error.

21.6 chunk — Read IFF chunked data

This module provides an interface for reading files that use EA IFF 85 chunks.[1] This format is used in at least the Audio Interchange File Format (AIFF/AIFF-C) and the Real Media File Format (RMFF). The WAVE audio file format is closely related and can also be read using this module.

A chunk has the following structure:

Offset	Length	Contents
0	4	Chunk ID
4	4	Size of chunk in big-endian byte order, not including the header
8	n	Data bytes, where n is the size given in the preceding field
8 + n	0 or 1	Pad byte needed if n is odd and chunk alignment is used

The ID is a 4-byte string which identifies the type of chunk.

[1] "EA IFF 85" Standard for Interchange Format Files, Jerry Morrison, Electronic Arts, January 1985.

The size field (a 32-bit value, encoded using big-endian byte order) gives the size of the chunk data, not including the 8-byte header.

Usually an IFF-type file consists of one or more chunks. The proposed usage of the Chunk class defined here is to instantiate an instance at the start of each chunk and read from the instance until it reaches the end, after which a new instance can be instantiated. At the end of the file, creating a new instance will fail with an EOFError exception.

class chunk.Chunk(file[, align, bigendian, inclheader])
Class which represents a chunk. The file argument is expected to be a file-like object. An instance of this class is specifically allowed. The only method that is needed is read(). If the methods seek() and tell() are present and don't raise an exception, they are also used. If these methods are present and raise an exception, they are expected to not have altered the object. If the optional argument align is true, chunks are assumed to be aligned on 2-byte boundaries. If align is false, no alignment is assumed. The default value is true. If the optional argument bigendian is false, the chunk size is assumed to be in little-endian order. This is needed for WAVE audio files. The default value is true. If the optional argument inclheader is true, the size given in the chunk header includes the size of the header. The default value is false.

A Chunk object supports the following methods:

getname()
Returns the name (ID) of the chunk. This is the first 4 bytes of the chunk.

getsize()
Returns the size of the chunk.

close()
Close and skip to the end of the chunk. This does not close the underlying file.

The remaining methods will raise IOError if called after the close() method has been called.

isatty()
Returns False.

seek(pos[, whence])
Set the chunk's current position. The whence argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file's end). There is no return value. If the underlying file does not allow seek, only forward seeks are allowed.

tell()
Return the current position into the chunk.

read([size])
Read at most size bytes from the chunk (less if the read hits the end of the chunk before obtaining size bytes). If the size argument is negative or omitted, read all data until the end of the chunk. The bytes are returned as a string object. An empty string is returned when the end of the chunk is encountered immediately.

skip()
Skip to the end of the chunk. All further calls to read() for the chunk will return `''`. If you are not interested in the contents of the chunk, this method should be called so that the file points to the start of the next chunk.

21.7 colorsys — Conversions between color systems

Source code: Lib/colorsys.py

The colorsys module defines bidirectional conversions of color values between colors expressed in the RGB (Red Green Blue) color space used in computer monitors and three other coordinate systems: YIQ, HLS (Hue Lightness Saturation) and HSV (Hue Saturation Value). Coordinates in all of these color spaces are floating point values. In the YIQ space, the Y coordinate is between 0 and 1, but the I and Q coordinates can be positive or negative. In all other spaces, the coordinates are all between 0 and 1.

See also:

More information about color spaces can be found at http://www.poynton.com/ColorFAQ.html and https://www.cambridgeincolour.com/tutorials/color-spaces.htm.

The colorsys module defines the following functions:

colorsys.rgb_to_yiq(r, g, b)
: Convert the color from RGB coordinates to YIQ coordinates.

colorsys.yiq_to_rgb(y, i, q)
: Convert the color from YIQ coordinates to RGB coordinates.

colorsys.rgb_to_hls(r, g, b)
: Convert the color from RGB coordinates to HLS coordinates.

colorsys.hls_to_rgb(h, l, s)
: Convert the color from HLS coordinates to RGB coordinates.

colorsys.rgb_to_hsv(r, g, b)
: Convert the color from RGB coordinates to HSV coordinates.

colorsys.hsv_to_rgb(h, s, v)
: Convert the color from HSV coordinates to RGB coordinates.

Example:

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

21.8 imghdr — Determine the type of an image

Source code: Lib/imghdr.py

The imghdr module determines the type of image contained in a file or byte stream.

The imghdr module defines the following function:

imghdr.what(filename[, h])
: Tests the image data contained in the file named by filename, and returns a string describing the image type. If optional h is provided, the filename is ignored and h is assumed to contain the byte stream to test.

The following image types are recognized, as listed below with the return value from what():

Value	Image format
'rgb'	SGI ImgLib Files
'gif'	GIF 87a and 89a Files
'pbm'	Portable Bitmap Files
'pgm'	Portable Graymap Files
'ppm'	Portable Pixmap Files
'tiff'	TIFF Files
'rast'	Sun Raster Files
'xbm'	X Bitmap Files
'jpeg'	JPEG data in JFIF or Exif formats
'bmp'	BMP files
'png'	Portable Network Graphics

New in version 2.5: Exif detection.

You can extend the list of file types imghdr can recognize by appending to this variable:

imghdr.tests
: A list of functions performing the individual tests. Each function takes two arguments: the byte-stream and an open file-like object. When what() is called with a byte-stream, the file-like object will be None.

  The test function should return a string describing the image type if the test succeeded, or None if it failed.

Example:

```
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

21.9 sndhdr — Determine type of sound file

Source code: Lib/sndhdr.py

The sndhdr provides utility functions which attempt to determine the type of sound data which is in a file. When these functions are able to determine what type of sound data is stored in a file, they return a tuple (type, sampling_rate, channels, frames, bits_per_sample). The value for type indicates the data type and will be one of the strings 'aifc', 'aiff', 'au', 'hcom', 'sndr', 'sndt', 'voc', 'wav', '8svx', 'sb', 'ub', or 'ul'. The sampling_rate will be either the actual value or 0 if unknown or difficult to decode. Similarly, channels will be either the number of channels or 0 if it cannot be determined or if the value is difficult to decode. The value for frames will be either the number of frames or -1. The last item in the tuple, bits_per_sample, will either be the sample size in bits or 'A' for A-LAW or 'U' for u-LAW.

sndhdr.what(filename)
: Determines the type of sound data stored in the file filename using whathdr(). If it succeeds, returns a tuple as described above, otherwise None is returned.

sndhdr.whathdr(filename)
: Determines the type of sound data stored in a file based on the file header. The name of the file is given by filename. This function returns a tuple as described above on success, or None.

21.10 ossaudiodev — Access to OSS-compatible audio devices

New in version 2.3.

This module allows you to access the OSS (Open Sound System) audio interface. OSS is available for a wide range of open-source and commercial Unices, and is the standard audio interface for Linux and recent versions of FreeBSD.

See also:

Open Sound System Programmer's Guide the official documentation for the OSS C API

The module defines a large number of constants supplied by the OSS device driver; see <sys/soundcard.h> on either Linux or FreeBSD for a listing.

ossaudiodev defines the following variables and functions:

exception ossaudiodev.OSSAudioError
: This exception is raised on certain errors. The argument is a string describing what went wrong.

  (If ossaudiodev receives an error from a system call such as open(), write(), or ioctl(), it raises IOError. Errors detected directly by ossaudiodev result in OSSAudioError.)

  (For backwards compatibility, the exception class is also available as ossaudiodev.error.)

ossaudiodev.open(mode)
ossaudiodev.open(device, mode)
: Open an audio device and return an OSS audio device object. This object supports many file-like methods, such as read(), write(), and fileno() (although there are subtle differences between conventional Unix read/write semantics and those of OSS audio devices). It also supports a number of audio-specific methods; see below for the complete list of methods.

  device is the audio device filename to use. If it is not specified, this module first looks in the environment variable AUDIODEV for a device to use. If not found, it falls back to /dev/dsp.

  mode is one of `'r'` for read-only (record) access, `'w'` for write-only (playback) access and `'rw'` for both. Since many sound cards only allow one process to have the recorder or player open at a time, it is a good idea to open the device only for the activity needed. Further, some sound cards are half-duplex: they can be opened for reading or writing, but not both at once.

  Note the unusual calling syntax: the first argument is optional, and the second is required. This is a historical artifact for compatibility with the older linuxaudiodev module which ossaudiodev supersedes.

ossaudiodev.openmixer([device])
: Open a mixer device and return an OSS mixer device object. device is the mixer device filename to use. If it is not specified, this module first looks in the environment variable MIXERDEV for a device to use. If not found, it falls back to /dev/mixer.

21.10.1 Audio Device Objects

Before you can write to or read from an audio device, you must call three methods in the correct order:

1. setfmt() to set the output format
2. channels() to set the number of channels
3. speed() to set the sample rate

Alternately, you can use the setparameters() method to set all three audio parameters at once. This is more convenient, but may not be as flexible in all cases.

The audio device objects returned by open() define the following methods and (read-only) attributes:

oss_audio_device.close()
Explicitly close the audio device. When you are done writing to or reading from an audio device, you should explicitly close it. A closed device cannot be used again.

oss_audio_device.fileno()
Return the file descriptor associated with the device.

oss_audio_device.read(size)
Read size bytes from the audio input and return them as a Python string. Unlike most Unix device drivers, OSS audio devices in blocking mode (the default) will block read() until the entire requested amount of data is available.

oss_audio_device.write(data)
Write the Python string data to the audio device and return the number of bytes written. If the audio device is in blocking mode (the default), the entire string is always written (again, this is different from usual Unix device semantics). If the device is in non-blocking mode, some data may not be written —see writeall().

oss_audio_device.writeall(data)
Write the entire Python string data to the audio device: waits until the audio device is able to accept data, writes as much data as it will accept, and repeats until data has been completely written. If the device is in blocking mode (the default), this has the same effect as write(); writeall() is only useful in non-blocking mode. Has no return value, since the amount of data written is always equal to the amount of data supplied.

The following methods each map to exactly one ioctl() system call. The correspondence is obvious: for example, setfmt() corresponds to the SNDCTL_DSP_SETFMT ioctl, and sync() to SNDCTL_DSP_SYNC (this can be useful when consulting the OSS documentation). If the underlying ioctl() fails, they all raise IOError.

oss_audio_device.nonblock()
Put the device into non-blocking mode. Once in non-blocking mode, there is no way to return it to blocking mode.

oss_audio_device.getfmts()
Return a bitmask of the audio output formats supported by the soundcard. Some of the formats supported by OSS are:

Format	Description
AFMT_MU_LAW	a logarithmic encoding (used by Sun .au files and /dev/audio)
AFMT_A_LAW	a logarithmic encoding
AFMT_IMA_ADPCM	a 4:1 compressed format defined by the Interactive Multimedia Association
AFMT_U8	Unsigned, 8-bit audio
AFMT_S16_LE	Signed, 16-bit audio, little-endian byte order (as used by Intel processors)
AFMT_S16_BE	Signed, 16-bit audio, big-endian byte order (as used by 68k, PowerPC, Sparc)
AFMT_S8	Signed, 8 bit audio
AFMT_U16_LE	Unsigned, 16-bit little-endian audio
AFMT_U16_BE	Unsigned, 16-bit big-endian audio

Consult the OSS documentation for a full list of audio formats, and note that most devices support only a subset of these formats. Some older devices only support AFMT_U8; the most common format used today is AFMT_S16_LE.

oss_audio_device.setfmt(format)
Try to set the current audio format to format—see getfmts() for a list. Returns the audio format that

the device was set to, which may not be the requested format. May also be used to return the current audio format—do this by passing an "audio format" of AFMT_QUERY.

oss_audio_device.channels(nchannels)
Set the number of output channels to nchannels. A value of 1 indicates monophonic sound, 2 stereophonic. Some devices may have more than 2 channels, and some high-end devices may not support mono. Returns the number of channels the device was set to.

oss_audio_device.speed(samplerate)
Try to set the audio sampling rate to samplerate samples per second. Returns the rate actually set. Most sound devices don't support arbitrary sampling rates. Common rates are:

Rate	Description
8000	default rate for /dev/audio
11025	speech recording
22050	
44100	CD quality audio (at 16 bits/sample and 2 channels)
96000	DVD quality audio (at 24 bits/sample)

oss_audio_device.sync()
Wait until the sound device has played every byte in its buffer. (This happens implicitly when the device is closed.) The OSS documentation recommends closing and re-opening the device rather than using sync().

oss_audio_device.reset()
Immediately stop playing or recording and return the device to a state where it can accept commands. The OSS documentation recommends closing and re-opening the device after calling reset().

oss_audio_device.post()
Tell the driver that there is likely to be a pause in the output, making it possible for the device to handle the pause more intelligently. You might use this after playing a spot sound effect, before waiting for user input, or before doing disk I/O.

The following convenience methods combine several ioctls, or one ioctl and some simple calculations.

oss_audio_device.setparameters(format, nchannels, samplerate[, strict=False])
Set the key audio sampling parameters—sample format, number of channels, and sampling rate—in one method call. format, nchannels, and samplerate should be as specified in the setfmt(), channels(), and speed() methods. If strict is true, setparameters() checks to see if each parameter was actually set to the requested value, and raises OSSAudioError if not. Returns a tuple (format, nchannels, samplerate) indicating the parameter values that were actually set by the device driver (i.e., the same as the return values of setfmt(), channels(), and speed()).

For example,

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

is equivalent to

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

oss_audio_device.bufsize()
Returns the size of the hardware buffer, in samples.

oss_audio_device.obufcount()
Returns the number of samples that are in the hardware buffer yet to be played.

oss_audio_device.obuffree()
Returns the number of samples that could be queued into the hardware buffer to be played without blocking.

Audio device objects also support several read-only attributes:

oss_audio_device.closed
Boolean indicating whether the device has been closed.

oss_audio_device.name
String containing the name of the device file.

oss_audio_device.mode
The I/O mode for the file, either "r", "rw", or "w".

21.10.2 Mixer Device Objects

The mixer object provides two file-like methods:

oss_mixer_device.close()
This method closes the open mixer device file. Any further attempts to use the mixer after this file is closed will raise an IOError.

oss_mixer_device.fileno()
Returns the file handle number of the open mixer device file.

The remaining methods are specific to audio mixing:

oss_mixer_device.controls()
This method returns a bitmask specifying the available mixer controls ("Control" being a specific mixable "channel", such as SOUND_MIXER_PCM or SOUND_MIXER_SYNTH). This bitmask indicates a subset of all available mixer controls—the SOUND_MIXER_* constants defined at module level. To determine if, for example, the current mixer object supports a PCM mixer, use the following Python code:

```
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

For most purposes, the SOUND_MIXER_VOLUME (master volume) and SOUND_MIXER_PCM controls should suffice—but code that uses the mixer should be flexible when it comes to choosing mixer controls. On the Gravis Ultrasound, for example, SOUND_MIXER_VOLUME does not exist.

oss_mixer_device.stereocontrols()
Returns a bitmask indicating stereo mixer controls. If a bit is set, the corresponding control is stereo; if it is unset, the control is either monophonic or not supported by the mixer (use in combination with controls() to determine which).

See the code example for the controls() function for an example of getting data from a bitmask.

oss_mixer_device.reccontrols()
Returns a bitmask specifying the mixer controls that may be used to record. See the code example for controls() for an example of reading from a bitmask.

oss_mixer_device.get(control)
Returns the volume of a given mixer control. The returned volume is a 2-tuple (left_volume, right_volume). Volumes are specified as numbers from 0 (silent) to 100 (full volume). If the control is monophonic, a 2-tuple is still returned, but both volumes are the same.

Raises OSSAudioError if an invalid control was is specified, or IOError if an unsupported control is specified.

oss_mixer_device.set(control, (left, right))
Sets the volume for a given mixer control to (left,right). left and right must be ints and between 0 (silent) and 100 (full volume). On success, the new volume is returned as a 2-tuple. Note that this may not be exactly the same as the volume specified, because of the limited resolution of some soundcard's mixers.

Raises OSSAudioError if an invalid mixer control was specified, or if the specified volumes were out-of-range.

oss_mixer_device.get_recsrc()
This method returns a bitmask indicating which control(s) are currently being used as a recording source.

oss_mixer_device.set_recsrc(bitmask)
Call this function to specify a recording source. Returns a bitmask indicating the new recording source (or sources) if successful; raises IOError if an invalid source was specified. To set the current recording source to the microphone input:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```

CHAPTER

TWENTYTWO

INTERNATIONALIZATION

The modules described in this chapter help you write software that is independent of language and locale by providing mechanisms for selecting a language to be used in program messages or by tailoring output to match local conventions.

The list of modules described in this chapter is:

22.1 gettext — Multilingual internationalization services

Source code: Lib/gettext.py

The gettext module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU gettext message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

Some hints on localizing your Python modules and applications are also given.

22.1.1 GNU gettext API

The gettext module defines the following API, which is very similar to the GNU gettext API. If you use this API you will affect the translation of your entire application globally. Often this is what you want if your application is monolingual, with the choice of language dependent on the locale of your user. If you are localizing a Python module, or if your application needs to switch languages on the fly, you probably want to use the class-based API instead.

gettext.bindtextdomain(domain[, localedir])

Bind the domain to the locale directory localedir. More concretely, gettext will look for binary .mo files for the given domain using the path (on Unix): localedir/language/LC_MESSAGES/domain.mo, where languages is searched for in the environment variables LANGUAGE, LC_ALL, LC_MESSAGES, and LANG respectively.

If localedir is omitted or None, then the current binding for domain is returned.[1]

[1] The default locale directory is system dependent; for example, on RedHat Linux it is /usr/share/locale, but on Solaris it is /usr/lib/locale. The gettext module does not try to support these system dependent defaults; instead its default is sys.prefix/share/locale. For this reason, it is always best to call bindtextdomain() with an explicit absolute path at the start of your application.

gettext.bind_textdomain_codeset(domain[, codeset])
: Bind the domain to codeset, changing the encoding of strings returned by the gettext() family of functions. If codeset is omitted, then the current binding is returned.

  New in version 2.4.

gettext.textdomain([domain])
: Change or query the current global domain. If domain is None, then the current global domain is returned, otherwise the global domain is set to domain, which is returned.

gettext.gettext(message)
: Return the localized translation of message, based on the current global domain, language, and locale directory. This function is usually aliased as _() in the local namespace (see examples below).

gettext.lgettext(message)
: Equivalent to gettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with bind_textdomain_codeset().

  New in version 2.4.

gettext.dgettext(domain, message)
: Like gettext(), but look the message up in the specified domain.

gettext.ldgettext(domain, message)
: Equivalent to dgettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with bind_textdomain_codeset().

  New in version 2.4.

gettext.ngettext(singular, plural, n)
: Like gettext(), but consider plural forms. If a translation is found, apply the plural formula to n, and return the resulting message (some languages have more than two plural forms). If no translation is found, return singular if n is 1; return plural otherwise.

  The Plural formula is taken from the catalog header. It is a C or Python expression that has a free variable n; the expression evaluates to the index of the plural in the catalog. See the GNU gettext documentation for the precise syntax to be used in .po files and the formulas for a variety of languages.

  New in version 2.3.

gettext.lngettext(singular, plural, n)
: Equivalent to ngettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with bind_textdomain_codeset().

  New in version 2.4.

gettext.dngettext(domain, singular, plural, n)
: Like ngettext(), but look the message up in the specified domain.

  New in version 2.3.

gettext.ldngettext(domain, singular, plural, n)
: Equivalent to dngettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with bind_textdomain_codeset().

  New in version 2.4.

Note that GNU gettext also defines a dcgettext() method, but this was deemed not useful and so it is currently unimplemented.

Here's an example of typical usage for this API:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print _('This is a translatable string.')
```

22.1.2 Class-based API

The class-based API of the gettext module gives you more flexibility and greater convenience than the GNU gettext API. It is the recommended way of localizing your Python applications and modules. gettext defines a "translations" class which implements the parsing of GNU .mo format files, and has methods for returning either standard 8-bit strings or Unicode strings. Instances of this "translations" class can also install themselves in the built-in namespace as the function _().

gettext.find(domain[, localedir[, languages[, all]]])
: This function implements the standard .mo file search algorithm. It takes a domain, identical to what textdomain() takes. Optional localedir is as in bindtextdomain() Optional languages is a list of strings, where each string is a language code.

  If localedir is not given, then the default system locale directory is used.[2] If languages is not given, then the following environment variables are searched: LANGUAGE, LC_ALL, LC_MESSAGES, and LANG. The first one returning a non-empty value is used for the languages variable. The environment variables should contain a colon separated list of languages, which will be split on the colon to produce the expected list of language code strings.

  find() then expands and normalizes the languages, and then iterates through them, searching for an existing file built of these components:

  localedir/language/LC_MESSAGES/domain.mo

  The first such file name that exists is returned by find(). If no such file is found, then None is returned. If all is given, it returns a list of all file names, in the order in which they appear in the languages list or the environment variables.

gettext.translation(domain[, localedir[, languages[, class_[, fallback[, codeset]]]]])
: Return a Translations instance based on the domain, localedir, and languages, which are first passed to find() to get a list of the associated .mo file paths. Instances with identical .mo file names are cached. The actual class instantiated is either class_ if provided, otherwise GNUTranslations. The class's constructor must take a single file object argument. If provided, codeset will change the charset used to encode translated strings.

  If multiple files are found, later files are used as fallbacks for earlier ones. To allow setting the fallback, copy.copy() is used to clone each translation object from the cache; the actual instance data is still shared with the cache.

  If no .mo file is found, this function raises IOError if fallback is false (which is the default), and returns a NullTranslations instance if fallback is true.

  Changed in version 2.4: Added the codeset parameter.

gettext.install(domain[, localedir[, unicode[, codeset[, names]]]])
: This installs the function _() in Python's builtins namespace, based on domain, localedir, and codeset which are passed to the function translation(). The unicode flag is passed to the resulting translation object's install() method.

[2] See the footnote for bindtextdomain() above.

For the names parameter, please see the description of the translation object's install() method.

As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the _() function, like this:

```
print _('This string will be translated.')
```

For convenience, you want the _() function to be installed in Python's builtins namespace, so it is easily accessible in all modules of your application.

Changed in version 2.4: Added the codeset parameter.

Changed in version 2.5: Added the names parameter.

The NullTranslations class

Translation classes are what actually implement the translation of original source file message strings to translated message strings. The base class used by all translation classes is NullTranslations; this provides the basic interface you can use to write your own specialized translation classes. Here are the methods of NullTranslations:

class gettext.NullTranslations([fp])
Takes an optional file object fp, which is ignored by the base class. Initializes "protected" instance variables _info and _charset which are set by derived classes, as well as _fallback, which is set through add_fallback(). It then calls self._parse(fp) if fp is not None.

_parse(fp)
No-op'd in the base class, this method takes file object fp, and reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

add_fallback(fallback)
Add fallback as the fallback object for the current translation object. A translation object should consult the fallback if it cannot provide a translation for a given message.

gettext(message)
If a fallback has been set, forward gettext() to the fallback. Otherwise, return the translated message. Overridden in derived classes.

lgettext(message)
If a fallback has been set, forward lgettext() to the fallback. Otherwise, return the translated message. Overridden in derived classes.

New in version 2.4.

ugettext(message)
If a fallback has been set, forward ugettext() to the fallback. Otherwise, return the translated message as a Unicode string. Overridden in derived classes.

ngettext(singular, plural, n)
If a fallback has been set, forward ngettext() to the fallback. Otherwise, return the translated message. Overridden in derived classes.

New in version 2.3.

lngettext(singular, plural, n)
If a fallback has been set, forward lngettext() to the fallback. Otherwise, return the translated message. Overridden in derived classes.

New in version 2.4.

ungettext(singular, plural, n)
: If a fallback has been set, forward ungettext() to the fallback. Otherwise, return the translated message as a Unicode string. Overridden in derived classes.

  New in version 2.3.

info()
: Return the "protected" _info variable.

charset()
: Return the "protected" _charset variable.

output_charset()
: Return the "protected" _output_charset variable, which defines the encoding used to return translated messages.

  New in version 2.4.

set_output_charset(charset)
: Change the "protected" _output_charset variable, which defines the encoding used to return translated messages.

  New in version 2.4.

install([unicode[, names]])
: If the unicode flag is false, this method installs self.gettext() into the built-in namespace, binding it to _. If unicode is true, it binds self.ugettext() instead. By default, unicode is false.

  If the names parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to _(). Supported names are 'gettext' (bound to self.gettext() or self.ugettext() according to the unicode flag), 'ngettext' (bound to self.ngettext() or self.ungettext() according to the unicode flag), 'lgettext' and 'lngettext'.

  Note that this is only one way, albeit the most convenient way, to make the _() function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install _(). Instead, they should use this code to make _() available to their module:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

  This puts _() only in the module's global namespace and so only affects calls within this module.

  Changed in version 2.5: Added the names parameter.

The GNUTranslations class

The gettext module provides one additional class derived from NullTranslations: GNUTranslations. This class overrides _parse() to enable reading GNU gettext format .mo files in both big-endian and little-endian format. It also coerces both message ids and message strings to Unicode.

GNUTranslations parses optional meta-data out of the translation catalog. It is convention with GNU gettext to include meta-data as the translation for the empty string. This meta-data is in RFC 822-style key: value pairs, and should contain the Project-Id-Version key. If the key Content-Type is found, then the charset property is used to initialize the "protected" _charset instance variable, defaulting to None if not found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding. The ugettext() method always returns a Unicode, while the gettext() returns an encoded 8-bit string. For the message id arguments of both methods, either Unicode strings or 8-bit strings containing only US-ASCII characters are acceptable. Note that the Unicode version

of the methods (i.e. ugettext() and ungettext()) are the recommended interface to use for internationalized Python programs.

The entire set of key/value pairs are placed into a dictionary and set as the "protected" _info instance variable.

If the .mo file's magic number is invalid, or if other problems occur while reading the file, instantiating a GNUTranslations class can raise IOError.

The following methods are overridden from the base class implementation:

GNUTranslations.gettext(message)
: Look up the message id in the catalog and return the corresponding message string, as an 8-bit string encoded with the catalog's charset encoding, if known. If there is no entry in the catalog for the message id, and a fallback has been set, the look up is forwarded to the fallback's gettext() method. Otherwise, the message id is returned.

GNUTranslations.lgettext(message)
: Equivalent to gettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with set_output_charset().

  New in version 2.4.

GNUTranslations.ugettext(message)
: Look up the message id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the message id, and a fallback has been set, the look up is forwarded to the fallback's ugettext() method. Otherwise, the message id is returned.

GNUTranslations.ngettext(singular, plural, n)
: Do a plural-forms lookup of a message id. singular is used as the message id for purposes of lookup in the catalog, while n is used to determine which plural form to use. The returned message string is an 8-bit string encoded with the catalog's charset encoding, if known.

  If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's ngettext() method. Otherwise, when n is 1 singular is returned, and plural is returned in all other cases.

  New in version 2.3.

GNUTranslations.lngettext(singular, plural, n)
: Equivalent to gettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with set_output_charset().

  New in version 2.4.

GNUTranslations.ungettext(singular, plural, n)
: Do a plural-forms lookup of a message id. singular is used as the message id for purposes of lookup in the catalog, while n is used to determine which plural form to use. The returned message string is a Unicode string.

  If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's ungettext() method. Otherwise, when n is 1 singular is returned, and plural is returned in all other cases.

  Here is an example:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ungettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

New in version 2.3.

Solaris message catalog support

The Solaris operating system defines its own binary .mo file format, but since no documentation can be found on this format, it is not supported at this time.

The Catalog constructor

GNOME uses a version of the gettext module by James Henstridge, but this version has a slightly different API. Its documented usage was:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print _('hello world')
```

For compatibility with this older module, the function Catalog() is an alias for the translation() function described above.

One difference between this module and Henstridge's: his catalog objects supported access through a mapping API, but this appears to be unused and so is not currently supported.

22.1.3 Internationalizing your programs and modules

Internationalization (I18N) refers to the operation by which a program is made aware of multiple languages. Localization (L10N) refers to the adaptation of your program, once internationalized, to the local language and cultural habits. In order to provide multilingual messages for your Python programs, you need to take the following steps:

1. prepare your program or module by specially marking translatable strings
2. run a suite of tools over your marked files to generate raw messages catalogs
3. create language specific translations of the message catalogs
4. use the gettext module so that message strings are properly translated

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in _('...') — that is, a call to the function _(). For example:

```
filename = 'mylog.txt'
message = _('writing a log message')
fp = open(filename, 'w')
fp.write(message)
fp.close()
```

In this example, the string 'writing a log message' is marked as a candidate for translation, while the strings 'mylog.txt' and 'w' are not.

The Python distribution comes with two tools which help you generate the message catalogs once you've prepared your source code. These may or may not be available from a binary distribution, but they can be found in a source distribution, in the Tools/i18n directory.

The pygettext[3] program scans all your Python source code looking for the strings you previously marked as translatable. It is similar to the GNU gettext program except that it understands all the intricacies of Python source code, but knows nothing about C or C++ source code. You don't need GNU gettext unless you're also going to be translating C code (such as C extension modules).

pygettext generates textual Uniforum-style human readable message catalog .pot files, essentially structured human readable files which contain every marked string in the source code, along with a placeholder for the translation strings. pygettext is a command line script that supports a similar command line interface as xgettext; for details on its use, run:

```
pygettext.py --help
```

Copies of these .pot files are then handed over to the individual human translators who write language-specific versions for every supported natural language. They send you back the filled in language-specific versions as a .po file. Using the msgfmt.py[4] program (in the Tools/i18n directory), you take the .po files from your translators and generate the machine-readable .mo binary catalog files. The .mo files are what the gettext module uses for the actual translation processing during run-time.

How you use the gettext module in your code depends on whether you are internationalizing a single module or your entire application. The next two sections will discuss each case.

Localizing your module

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU gettext API but instead the class-based API.

Let's say your module is called "spam" and the module's various natural language translation .mo files reside in /usr/share/locale in GNU gettext format. Here's what you would put at the top of your module:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.lgettext
```

If your translators were providing you with Unicode strings in their .po files, you'd instead do:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.ugettext
```

Localizing your application

If you are localizing your application, you can install the _() function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use _('...') without having to explicitly install it in each file.

In the simple case then, you need only add the following bit of code to the main driver file of your application:

```
import gettext
gettext.install('myapplication')
```

If you need to set the locale directory or the unicode flag, you can pass these into the install() function:

[3] François Pinard has written a program called xpot which does a similar job. It is available as part of his po-utils package.

[4] msgfmt.py is binary compatible with GNU msgfmt except that it provides a simpler, all-Python implementation. With this and pygettext.py, you generally won't need to install the GNU gettext package to internationalize your Python applications.

```
import gettext
gettext.install('myapplication', '/usr/share/locale', unicode=1)
```

Changing languages on the fly

If your program needs to support many languages at the same time, you may want to create multiple translation instances and then switch between them explicitly, like so:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

Deferred translations

In most coding situations, strings are translated where they are coded. Occasionally however, you need to mark strings for translation, but defer actual translation until later. A classic example is:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
# ...
for a in animals:
    print a
```

Here, you want to mark the strings in the animals list as being translatable, but you don't actually want to translate them until they are printed.

Here is one way you can handle this situation:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print _(a)
```

This works because the dummy definition of _() simply returns the string unchanged. And this dummy definition will temporarily override any definition of _() in the built-in namespace (until the del command). Take care, though if you have a previous definition of _() in the local namespace.

Note that the second use of _() will not identify "a" as being translatable to the pygettext program, since it is not a string.

Another way to handle this is with the following example:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print _(a)
```

In this case, you are marking translatable strings with the function N_(),[5] which won't conflict with any definition of _(). However, you will need to teach your message extraction program to look for translatable strings marked with N_(). pygettext and xpot both support this through the use of command line switches.

gettext() vs. lgettext()

In Python 2.4 the lgettext() family of functions were introduced. The intention of these functions is to provide an alternative which is more compliant with the current implementation of GNU gettext. Unlike gettext(), which returns strings encoded with the same codeset used in the translation file, lgettext() will return strings encoded with the preferred system encoding, as returned by locale.getpreferredencoding(). Also notice that Python 2.4 introduces new functions to explicitly choose the codeset used in translated strings. If a codeset is explicitly set, even lgettext() will return translated strings in the requested codeset, as would be expected in the GNU gettext implementation.

22.1.4 Acknowledgements

The following people contributed code, feedback, design suggestions, previous implementations, and valuable experience to the creation of this module:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

[5] The choice of N_() here is totally arbitrary; it could have just as easily been MarkThisStringForTranslation().

22.2 locale — Internationalization services

The locale module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

The locale module is implemented on top of the _locale module, which in turn uses an ANSI C locale implementation if available.

The locale module defines the following exception and functions:

exception locale.Error
: Exception raised when the locale passed to setlocale() is not recognized.

locale.setlocale(category[, locale])
: If locale is given and not None, setlocale() modifies the locale setting for the category. The available categories are listed in the data description below. locale may be a string, or an iterable of two strings (language code and encoding). If it's an iterable, it's converted to a locale name using the locale aliasing engine. An empty string specifies the user's default settings. If the modification of the locale fails, the exception Error is raised. If successful, the new locale setting is returned.

  If locale is omitted or None, the current setting for category is returned.

  setlocale() is not thread-safe on most systems. Applications typically start with a call of

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

  This sets the locale for all categories to the user's default setting (typically specified in the LANG environment variable). If the locale is not changed thereafter, using multithreading should not cause problems.

  Changed in version 2.0: Added support for iterable values of the locale parameter.

locale.localeconv()
: Returns the database of the local conventions as a dictionary. This dictionary has the following strings as keys:

Category	Key	Meaning
LC_NUMERIC	'decimal_point'	Decimal point character.
	'grouping'	Sequence of numbers specifying which relative positions the 'thousands_sep' is expected. If the sequence is terminated with CHAR_MAX, no further grouping is performed. If the sequence terminates with a 0, the last group size is repeatedly used.
	'thousands_sep'	Character used between groups.
LC_MONETARY	'int_curr_symbol'	International currency symbol.
	'currency_symbol'	Local currency symbol.
	'p_cs_precedes/n_cs_precedes'	Whether the currency symbol precedes the value (for positive resp. negative values).
	'p_sep_by_space/n_sep_by_space'	Whether the currency symbol is separated from the value by a space (for positive resp. negative values).
	'mon_decimal_point'	Decimal point used for monetary values.
	'frac_digits'	Number of fractional digits used in local formatting of monetary values.
	'int_frac_digits'	Number of fractional digits used in international formatting of monetary values.
	'mon_thousands_sep'	Group separator used for monetary values.
	'mon_grouping'	Equivalent to 'grouping', used for monetary values.
	'positive_sign'	Symbol used to annotate a positive monetary value.
	'negative_sign'	Symbol used to annotate a negative monetary value.
	'p_sign_posn/n_sign_posn'	The position of the sign (for positive resp. negative values), see below.

All numeric values can be set to CHAR_MAX to indicate that there is no value specified in this locale.

The possible values for 'p_sign_posn' and 'n_sign_posn' are given below.

Value	Explanation
0	Currency and value are surrounded by parentheses.
1	The sign should precede the value and currency symbol.
2	The sign should follow the value and currency symbol.
3	The sign should immediately precede the value.
4	The sign should immediately follow the value.
CHAR_MAX	Nothing is specified in this locale.

locale.nl_langinfo(option)
Return some locale-specific information as a string. This function is not available on all systems, and the set of possible options might also vary across platforms. The possible argument values are numbers, for which symbolic constants are available in the locale module.

The nl_langinfo() function accepts one of the following keys. Most descriptions are taken from the corresponding description in the GNU C library.

locale.CODESET
Get a string with the name of the character encoding used in the selected locale.

locale.D_T_FMT
Get a string that can be used as a format string for time.strftime() to represent date and time in a locale-specific way.

locale.D_FMT
Get a string that can be used as a format string for time.strftime() to represent a date in a locale-specific way.

locale.T_FMT
Get a string that can be used as a format string for time.strftime() to represent a time in a locale-specific way.

locale.T_FMT_AMPM
Get a format string for time.strftime() to represent time in the am/pm format.

DAY_1 ... DAY_7
Get the name of the n-th day of the week.

Note: This follows the US convention of DAY_1 being Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.

ABDAY_1 ... ABDAY_7
Get the abbreviated name of the n-th day of the week.

MON_1 ... MON_12
Get the name of the n-th month.

ABMON_1 ... ABMON_12
Get the abbreviated name of the n-th month.

locale.RADIXCHAR
Get the radix character (decimal dot, decimal comma, etc.).

locale.THOUSEP
Get the separator character for thousands (groups of three digits).

locale.YESEXPR
Get a regular expression that can be used with the regex function to recognize a positive response to a yes/no question.

Note: The expression is in the syntax suitable for the regex() function from the C library, which might differ from the syntax used in re.

---

locale.NOEXPR
Get a regular expression that can be used with the regex(3) function to recognize a negative response to a yes/no question.

locale.CRNCYSTR
: Get the currency symbol, preceded by "-" if the symbol should appear before the value, "+" if the symbol should appear after the value, or "." if the symbol should replace the radix character.

locale.ERA
: Get a string that represents the era used in the current locale.

  Most locales do not define this value. An example of a locale which does define this value is the Japanese one. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor's reign.

  Normally it should not be necessary to use this value directly. Specifying the E modifier in their format strings causes the time.strftime() function to use this information. The format of the returned string is not specified, and therefore you should not assume knowledge of it on different systems.

locale.ERA_D_T_FMT
: Get a format string for time.strftime() to represent date and time in a locale-specific era-based way.

locale.ERA_D_FMT
: Get a format string for time.strftime() to represent a date in a locale-specific era-based way.

locale.ERA_T_FMT
: Get a format string for time.strftime() to represent a time in a locale-specific era-based way.

locale.ALT_DIGITS
: Get a representation of up to 100 values used to represent the values 0 to 99.

locale.getdefaultlocale([envvars])
: Tries to determine the default locale settings and returns them as a tuple of the form (language code, encoding).

  According to POSIX, a program which has not called setlocale(LC_ALL, '') runs using the portable 'C' locale. Calling setlocale(LC_ALL, '') lets it use the default locale as defined by the LANG variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

  To maintain compatibility with other platforms, not only the LANG variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. envvars defaults to the search path used in GNU gettext; it must always contain the variable name LANG. The GNU gettext search path contains 'LANGUAGE', 'LC_ALL', 'LC_CTYPE', and 'LANG', in that order.

  Except for the code 'C', the language code corresponds to RFC 1766. language code and encoding may be None if their values cannot be determined.

  New in version 2.0.

locale.getlocale([category])
: Returns the current setting for the given locale category as sequence containing language code, encoding. category may be one of the LC_* values except LC_ALL. It defaults to LC_CTYPE.

  Except for the code 'C', the language code corresponds to RFC 1766. language code and encoding may be None if their values cannot be determined.

  New in version 2.0.

locale.getpreferredencoding([do_setlocale])
: Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

On some systems, it is necessary to invoke setlocale() to obtain the user preferences, so this function is not thread-safe. If invoking setlocale is not necessary or desired, do_setlocale should be set to False.

New in version 2.3.

locale.normalize(localename)
Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with setlocale(). If normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like setlocale().

New in version 2.0.

locale.resetlocale([category])
Sets the locale for category to the default setting.

The default setting is determined by calling getdefaultlocale(). category defaults to LC_ALL.

New in version 2.0.

locale.strcoll(string1, string2)
Compares two strings according to the current LC_COLLATE setting. As any other compare function, returns a negative, or a positive value, or 0, depending on whether string1 collates before or after string2 or is equal to it.

locale.strxfrm(string)
Transforms a string to one that can be used for the built-in function cmp(), and still returns locale-aware results. This function can be used when the same string is compared repeatedly, e.g. when collating a sequence of strings.

locale.format(format, val[, grouping[, monetary]])
Formats a number val according to the current LC_NUMERIC setting. The format follows the conventions of the % operator. For floating point values, the decimal point is modified if appropriate. If grouping is true, also takes the grouping into account.

If monetary is true, the conversion uses monetary thousands separator and grouping strings.

Please note that this function will only work for exactly one %char specifier. For whole format strings, use format_string().

Changed in version 2.5: Added the monetary parameter.

locale.format_string(format, val[, grouping])
Processes formatting specifiers as in format % val, but takes the current locale settings into account.

New in version 2.5.

locale.currency(val[, symbol[, grouping[, international]]])
Formats a number val according to the current LC_MONETARY settings.

The returned string includes the currency symbol if symbol is true, which is the default. If grouping is true (which is not the default), grouping is done with the value. If international is true (which is not the default), the international currency symbol is used.

Note that this function will not work with the 'C' locale, so you have to set a locale via setlocale() first.

New in version 2.5.

locale.str(float)
Formats a floating point number using the same format as the built-in function str(float), but takes the decimal point into account.

locale.atof(string)
Converts a string to a floating point number, following the LC_NUMERIC settings.

locale.atoi(string)
Converts a string to an integer, following the LC_NUMERIC conventions.

locale.LC_CTYPE
Locale category for the character type functions. Depending on the settings of this category, the functions of module string dealing with case change their behaviour.

locale.LC_COLLATE
Locale category for sorting strings. The functions strcoll() and strxfrm() of the locale module are affected.

locale.LC_TIME
Locale category for the formatting of time. The function time.strftime() follows these conventions.

locale.LC_MONETARY
Locale category for formatting of monetary values. The available options are available from the localeconv() function.

locale.LC_MESSAGES
Locale category for message display. Python currently does not support application specific locale-aware messages. Messages displayed by the operating system, like those returned by os.strerror() might be affected by this category.

locale.LC_NUMERIC
Locale category for formatting numbers. The functions format(), atoi(), atof() and str() of the locale module are affected by that category. All other numeric formatting operations are not affected.

locale.LC_ALL
Combination of all locale settings. If this flag is used when the locale is changed, setting the locale for all categories is attempted. If that fails for any category, no category is changed at all. When the locale is retrieved using this flag, a string indicating the setting for all categories is returned. This string can be later used to restore the settings.

locale.CHAR_MAX
This is a symbolic constant used for different values returned by localeconv().

Example:

```
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')  # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')  # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)  # restore saved locale
```

22.2.1 Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementation are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the C locale, no matter what the user's preferred locale is. The program must explicitly say that it wants the user's preferred locale settings by calling setlocale(LC_ALL, '').

It is generally a bad idea to call setlocale() in some library routine, since as a side effect it affects the entire program. Saving and restoring it is almost as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (such as string.lower(), or certain formats used with time.strftime()), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-C locale settings.

The case conversion functions in the string module are affected by the locale settings. When a call to the setlocale() function changes the LC_CTYPE settings, the variables string.lowercase, string.uppercase and string.letters are recalculated. Note that code that uses these variable through 'from ... import ...', e.g. from string import letters, is not affected by subsequent setlocale() calls.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: atof(), atoi(), format(), str().

22.2.2 For extension writers and programs that embed Python

Extension modules should never call setlocale(), except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is C).

When Python code uses the locale module to change the locale, this also affects the embedding application. If the embedding application doesn't want this to happen, it should remove the _locale extension module (which does all the work) from the table of built-in modules in the config.c file, and make sure that the _locale module is not accessible as a shared library.

22.2.3 Access to message catalogs

locale.gettext(msg)

locale.dgettext(domain, msg)

locale.dcgettext(domain, msg, category)

locale.textdomain(domain)

locale.bindtextdomain(domain, dir)

The locale module exposes the C library's gettext interface on systems that provide this interface. It consists of the functions gettext(), dgettext(), dcgettext(), textdomain(), bindtextdomain(), and bind_textdomain_codeset(). These are similar to the same functions in the gettext module, but use the C library's binary format for message catalogs, and the C library's search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use gettext instead. A known exception to this rule are applications that link with additional C libraries which internally invoke gettext() or dcgettext(). For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.

CHAPTER

TWENTYTHREE

PROGRAM FRAMEWORKS

The modules described in this chapter are frameworks that will largely dictate the structure of your program. Currently the modules described here are all oriented toward writing command-line interfaces.

The full list of modules described in this chapter is:

23.1 cmd — Support for line-oriented command interpreters

Source code: Lib/cmd.py

The Cmd class provides a simple framework for writing line-oriented command interpreters. These are often useful for test harnesses, administrative tools, and prototypes that will later be wrapped in a more sophisticated interface.

class cmd.Cmd([completekey[, stdin[, stdout]]])

A Cmd instance or subclass instance is a line-oriented interpreter framework. There is no good reason to instantiate Cmd itself; rather, it's useful as a superclass of an interpreter class you define yourself in order to inherit Cmd's methods and encapsulate action methods.

The optional argument completekey is the readline name of a completion key; it defaults to Tab. If completekey is not None and readline is available, command completion is done automatically.

The optional arguments stdin and stdout specify the input and output file objects that the Cmd instance or subclass instance will use for input and output. If not specified, they will default to sys.stdin and sys.stdout.

If you want a given stdin to be used, make sure to set the instance's use_rawinput attribute to False, otherwise stdin will be ignored.

Changed in version 2.3: The stdin and stdout parameters were added.

23.1.1 Cmd Objects

A Cmd instance has the following methods:

Cmd.cmdloop([intro])

Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action methods, passing them the remainder of the line as argument.

The optional argument is a banner or intro string to be issued before the first prompt (this overrides the intro class attribute).

If the readline module is loaded, input will automatically inherit bash-like history-list editing (e.g. Control-P scrolls back to the last command, Control-N forward to the next one, Control-F moves the cursor to the right non-destructively, Control-B moves the cursor to the left non-destructively, etc.).

An end-of-file on input is passed back as the string `'EOF'`.

An interpreter instance will recognize a command name foo if and only if it has a method do_foo(). As a special case, a line beginning with the character `'?'` is dispatched to the method do_help(). As another special case, a line beginning with the character `'!'` is dispatched to the method do_shell() (if such a method is defined).

This method will return when the postcmd() method returns a true value. The stop argument to postcmd() is the return value from the command's corresponding do_*() method.

If completion is enabled, completing commands will be done automatically, and completing of commands args is done by calling complete_foo() with arguments text, line, begidx, and endidx. text is the string prefix we are attempting to match: all returned matches must begin with it. line is the current input line with leading whitespace removed, begidx and endidx are the beginning and ending indexes of the prefix text, which could be used to provide different completion depending upon which position the argument is in.

All subclasses of Cmd inherit a predefined do_help(). This method, called with an argument `'bar'`, invokes the corresponding method help_bar(), and if that is not present, prints the docstring of do_bar(), if available. With no argument, do_help() lists all available help topics (that is, all commands with corresponding help_*() methods or commands that have docstrings), and also lists any undocumented commands.

Cmd.onecmd(str)
: Interpret the argument as though it had been typed in response to the prompt. This may be overridden, but should not normally need to be; see the precmd() and postcmd() methods for useful execution hooks. The return value is a flag indicating whether interpretation of commands by the interpreter should stop. If there is a do_*() method for the command str, the return value of that method is returned, otherwise the return value from the default() method is returned.

Cmd.emptyline()
: Method called when an empty line is entered in response to the prompt. If this method is not overridden, it repeats the last nonempty command entered.

Cmd.default(line)
: Method called on an input line when the command prefix is not recognized. If this method is not overridden, it prints an error message and returns.

Cmd.completedefault(text, line, begidx, endidx)
: Method called to complete an input line when no command-specific complete_*() method is available. By default, it returns an empty list.

Cmd.precmd(line)
: Hook method executed just before the command line line is interpreted, but after the input prompt is generated and issued. This method is a stub in Cmd; it exists to be overridden by subclasses. The return value is used as the command which will be executed by the onecmd() method; the precmd() implementation may re-write the command or simply return line unchanged.

Cmd.postcmd(stop, line)
: Hook method executed just after a command dispatch is finished. This method is a stub in Cmd; it exists to be overridden by subclasses. line is the command line which was executed, and stop is a flag which indicates whether execution will be terminated after the call to postcmd(); this will be the return value of the onecmd() method. The return value of this method will be used as the new value for the internal flag which corresponds to stop; returning false will cause interpretation to continue.

Cmd.preloop()
: Hook method executed once when cmdloop() is called. This method is a stub in Cmd; it exists to be overridden by subclasses.

Cmd.postloop()
: Hook method executed once when cmdloop() is about to return. This method is a stub in Cmd; it exists to be overridden by subclasses.

Instances of Cmd subclasses have some public instance variables:

Cmd.prompt
: The prompt issued to solicit input.

Cmd.identchars
: The string of characters accepted for the command prefix.

Cmd.lastcmd
: The last nonempty command prefix seen.

Cmd.cmdqueue
: A list of queued input lines. The cmdqueue list is checked in cmdloop() when new input is needed; if it is nonempty, its elements will be processed in order, as if entered at the prompt.

Cmd.intro
: A string to issue as an intro or banner. May be overridden by giving the cmdloop() method an argument.

Cmd.doc_header
: The header to issue if the help output has a section for documented commands.

Cmd.misc_header
: The header to issue if the help output has a section for miscellaneous help topics (that is, there are help_*() methods without corresponding do_*() methods).

Cmd.undoc_header
: The header to issue if the help output has a section for undocumented commands (that is, there are do_*() methods without corresponding help_*() methods).

Cmd.ruler
: The character used to draw separator lines under the help-message headers. If empty, no ruler line is drawn. It defaults to '='.

Cmd.use_rawinput
: A flag, defaulting to true. If true, cmdloop() uses raw_input() to display a prompt and read the next command; if false, sys.stdout.write() and sys.stdin.readline() are used. (This means that by importing readline, on systems that support it, the interpreter will automatically support Emacs-like line editing and command-history keystrokes.)

23.2 shlex — Simple lexical analysis

New in version 1.5.2.

Source code: Lib/shlex.py

The shlex class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

Prior to Python 2.7.3, this module did not support Unicode input.

The shlex module defines the following functions:

shlex.split(s[, comments[, posix]])
Split the string s using shell-like syntax. If comments is False (the default), the parsing of comments in the given string will be disabled (setting the commenters attribute of the shlex instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the posix argument is false.

New in version 2.3.

Changed in version 2.6: Added the posix parameter.

Note: Since the split() function instantiates a shlex instance, passing None for s will read the string to split from standard input.

The shlex module defines the following class:

class shlex.shlex([instream[, infile[, posix]]])
A shlex instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with read() and readline() methods, or a string (strings are accepted since Python 2.3). If no argument is given, input will be taken from sys.stdin. The second optional argument is a filename string, which sets the initial value of the infile attribute. If the instream argument is omitted or equal to sys.stdin, this second argument defaults to "stdin". The posix argument was introduced in Python 2.3, and defines the operational mode. When posix is not true (default), the shlex instance will operate in compatibility mode. When operating in POSIX mode, shlex will try to be as close as possible to the POSIX shell parsing rules.

See also:

Module ConfigParser Parser for configuration files similar to the Windows .ini files.

23.2.1 shlex Objects

A shlex instance has the following methods:

shlex.get_token()
Return a token. If tokens have been stacked using push_token(), pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, eof is returned (the empty string (`''`) in non-POSIX mode, and None in POSIX mode).

shlex.push_token(str)
Push the argument onto the token stack.

shlex.read_token()
Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

shlex.sourcehook(filename)
When shlex detects a source request (see source below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as sys.stdin), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles #include "file.h").

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with open() called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding 'close' hook, but a shlex instance will call the close() method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the push_source() and pop_source() methods.

shlex.push_source(stream[, filename])
Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the sourcehook() method.

New in version 2.1.

shlex.pop_source()
Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

New in version 2.1.

shlex.error_leader([file[, line]])
This method generates an error message leader in the format of a Unix C compiler error label; the format is '"%s", line %d: ', where the %s is replaced with the name of the current source file and the %d with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage shlex users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of shlex subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

shlex.commenters
The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just '#' by default.

shlex.wordchars
The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore.

shlex.whitespace
Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

shlex.escape
Characters that will be considered as escape. This will be only used in POSIX mode, and includes just '\' by default.

New in version 2.3.

shlex.quotes
Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

shlex.escapedquotes
Characters in quotes that will interpret escape characters defined in escape. This is only used in POSIX mode, and includes just '"' by default.

New in version 2.3.

shlex.whitespace_split
: If True, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with shlex, getting tokens in a similar way to shell arguments.

  New in version 2.3.

shlex.infile
: The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

shlex.instream
: The input stream from which this shlex instance is reading characters.

shlex.source
: This attribute is None by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the source keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the close() method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

shlex.debug
: If this attribute is numeric and 1 or more, a shlex instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

shlex.lineno
: Source line number (count of newlines seen so far plus one).

shlex.token
: The token buffer. It may be useful to examine this when catching exceptions.

shlex.eof
: Token used to determine end of file. This will be set to the empty string (''), in non-POSIX mode, and to None in POSIX mode.

  New in version 2.3.

23.2.2 Parsing Rules

When operating in non-POSIX mode, shlex will try to obey to the following rules.

- Quote characters are not recognized within words (Do"Not"Separate is parsed as the single word Do"Not"Separate);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words ("Do"Separate is parsed as "Do" and Separate);
- If whitespace_split is False, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is True, shlex will only split words in whitespaces;
- EOF is signaled with an empty string ('');
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, shlex will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words ("Do"Not"Separate" is parsed as the single word DoNotSeparate);
- Non-quoted escape characters (e.g. '\') preserve the literal value of the next character that follows;

- Enclosing characters in quotes which are not part of escapedquotes (e.g. `"'"`) preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of escapedquotes (e.g. `'"'`) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in escape. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.
- EOF is signaled with a None value;
- Quoted empty strings (`''`) are allowed;

CHAPTER

TWENTYFOUR

GRAPHICAL USER INTERFACES WITH TK

Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the Tkinter module, and its extensions, the Tix and the ttk modules.

The Tkinter module is a thin object-oriented layer on top of Tcl/Tk. To use Tkinter, you don't need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. Tkinter is a set of wrappers that implement the Tk widgets as Python classes. In addition, the internal module _tkinter provides a threadsafe mechanism which allows Python and Tcl to interact.

Tkinter's chief virtues are that it is fast, and that it usually comes bundled with Python. Although its standard documentation is weak, good material is available, which includes: references, tutorials, a book and others. Tkinter is also famous for having an outdated look and feel, which has been vastly improved in Tk 8.5. Nevertheless, there are many other GUI libraries that you could be interested in. For more information about alternatives, see the Other Graphical User Interface Packages section.

24.1 Tkinter — Python interface to Tcl/Tk

The Tkinter module ("Tk interface") is the standard Python interface to the Tk GUI toolkit. Both Tk and Tkinter are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.)

Running python -m Tkinter from the command line should open a window demonstrating a simple Tk interface, letting you know that Tkinter is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

Note: Tkinter has been renamed to tkinter in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

See also:

Tkinter documentation:

Python Tkinter Resources The Python Tkinter Topic Guide provides a great deal of information on using Tk from Python and links to other sources of information on Tk.

TKDocs Extensive tutorial plus friendlier widget pages for some of the widgets.

Tkinter reference: a GUI for Python On-line reference material.

Tkinter docs from effbot Online reference for tkinter supported by effbot.org.

Programming Python Book by Mark Lutz, has excellent coverage of Tkinter.

Modern Tkinter for Busy Python Developers Book by Mark Rozerman about building attractive and modern graphical user interfaces with Python and Tkinter.

Python and Tkinter Programming Book by John Grayson (ISBN 1-884777-81-3).

Tcl/Tk documentation:

Tk commands Most commands are available as Tkinter or Tkinter.ttk classes. Change '8.6' to match the version of your Tcl/Tk installation.

Tcl/Tk recent man pages Recent Tcl/Tk manuals on www.tcl.tk.

ActiveState Tcl Home Page The Tk/Tcl development is largely taking place at ActiveState.

Tcl and the Tk Toolkit Book by John Ousterhout, the inventor of Tcl.

Practical Programming in Tcl and Tk Brent Welch's encyclopedic book.

24.1.1 Tkinter Modules

Most of the time, the Tkinter module is all you really need, but a number of additional modules are available as well. The Tk interface is located in a binary module named _tkinter. This module contains the low-level interface to Tk, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

In addition to the Tk interface module, Tkinter includes a number of Python modules. The two most important modules are the Tkinter module itself, and a module called Tkconstants. The former automatically imports the latter, so to use Tkinter, all you need to do is to import one module:

```
import Tkinter
```

Or, more often:

```
from Tkinter import *
```

class Tkinter.Tk(screenName=None, baseName=None, className='Tk', useTk=1)
The Tk class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an application. Each instance has its own associated Tcl interpreter.

Changed in version 2.4: The useTk parameter was added.

Tkinter.Tcl(screenName=None, baseName=None, className='Tk', useTk=0)
The Tcl() function is a factory function which creates an object much like that created by the Tk class, except that it does not initialize the Tk subsystem. This is most often useful when driving the Tcl interpreter in an environment where one doesn't want to create extraneous toplevel windows, or where one cannot (such as Unix/Linux systems without an X server). An object created by the Tcl() object can have a Toplevel window created (and the Tk subsystem initialized) by calling its loadtk() method.

New in version 2.4.

Other modules that provide Tk support include:

ScrolledText Text widget with a vertical scroll bar built in.

tkColorChooser Dialog to let the user choose a color.

tkCommonDialog Base class for the dialogs defined in the other modules listed here.

tkFileDialog Common dialogs to allow the user to specify a file to open or save.

tkFont Utilities to help work with fonts.

tkMessageBox Access to standard Tk dialog boxes.

tkSimpleDialog Basic dialogs and convenience functions.

Tkdnd Drag-and-drop support for Tkinter. This is experimental and should become deprecated when it is replaced with the Tk DND.

turtle Turtle graphics in a Tk window.

These have been renamed as well in Python 3; they were all made submodules of the new tkinter package.

24.1.2 Tkinter Life Preserver

This section is not designed to be an exhaustive tutorial on either Tk or Tkinter. Rather, it is intended as a stop gap, providing some introductory orientation on the system.

Credits:

- Tkinter was written by Steen Lumholt and Guido van Rossum.
- Tk was written by John Ousterhout while at Berkeley.
- This Life Preserver was written by Matt Conway at the University of Virginia.
- The html rendering, and some liberal editing, was produced from a FrameMaker version by Ken Manheimer.
- Fredrik Lundh elaborated and revised the class interface descriptions, to get them current with Tk 4.2.
- Mike Clarkson converted the documentation to LaTeX, and compiled the User Interface chapter of the reference manual.

How To Use This Section

This section is designed in two parts: the first half (roughly) covers background material, while the second half can be taken to the keyboard as a handy reference.

When trying to answer questions of the form "how do I do blah", it is often best to find out how to do "blah" in straight Tk, and then convert this back into the corresponding Tkinter call. Python programmers can often guess at the correct Python command by looking at the Tk documentation. This means that in order to use Tkinter, you will have to know a little bit about Tk. This document can't fulfill that role, so the best we can do is point you to the best documentation that exists. Here are some hints:

- The authors strongly suggest getting a copy of the Tk man pages. Specifically, the man pages in the mann directory are most useful. The man3 man pages describe the C interface to the Tk library and thus are not especially helpful for script writers.
- Addison-Wesley publishes a book called Tcl and the Tk Toolkit by John Ousterhout (ISBN 0-201-63337-X) which is a good introduction to Tcl and Tk for the novice. The book is not exhaustive, and for many details it defers to the man pages.
- Tkinter.py is a last resort for most, but can be a good place to go when nothing else makes sense.

A Simple Hello World Program

```
from Tkinter import *

class Application(Frame):
    def say_hi(self):
        print "hi there, everyone!"
```

(continues on next page)

(continued from previous page)

```
    def createWidgets(self):
        self.QUIT = Button(self)
        self.QUIT["text"] = "QUIT"
        self.QUIT["fg"]   = "red"
        self.QUIT["command"] =  self.quit

        self.QUIT.pack({"side": "left"})

        self.hi_there = Button(self)
        self.hi_there["text"] = "Hello",
        self.hi_there["command"] = self.say_hi

        self.hi_there.pack({"side": "left"})

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

root = Tk()
app = Application(master=root)
app.mainloop()
root.destroy()
```

24.1.3 A (Very) Quick Look at Tcl/Tk

The class hierarchy looks complicated, but in actual practice, application programmers almost always refer to the classes at the very bottom of the hierarchy.

Notes:

- These classes are provided for the purposes of organizing certain functions under one namespace. They aren't meant to be instantiated independently.
- The Tk class is meant to be instantiated only once in an application. Application programmers need not instantiate one explicitly, the system creates one whenever any of the other classes are instantiated.
- The Widget class is not meant to be instantiated, it is meant only for subclassing to make "real" widgets (in C++, this is called an 'abstract class').

To make use of this reference material, there will be times when you will need to know how to read short passages of Tk and how to identify the various parts of a Tk command. (See section Mapping Basic Tk into Tkinter for the Tkinter equivalents of what's below.)

Tk scripts are Tcl programs. Like all Tcl programs, Tk scripts are just lists of tokens separated by spaces. A Tk widget is just its class, the options that help configure it, and the actions that make it do useful things.

To make a widget in Tk, the command is always of the form:

```
classCommand newPathname options
```

classCommand denotes which kind of widget to make (a button, a label, a menu...)

newPathname is the new name for this widget. All names in Tk must be unique. To help enforce this, widgets in Tk are named with pathnames, just like files in a file system. The top level widget, the root, is called . (period) and children are delimited by more periods. For example, .myApp.controlPanel.okButton might be the name of a widget.

options configure the widget's appearance and in some cases, its behavior. The options come in the form of a list of flags and values. Flags are preceded by a '-', like Unix shell command flags, and values are put in quotes if they are more than one word.

For example:

```
button   .fred   -fg red -text "hi there"
   ^       ^     \_____________________/
   |       |               |
 class    new            options
command  widget  (-opt val -opt val ...)
```

Once created, the pathname to the widget becomes a new command. This new widget command is the programmer's handle for getting the new widget to perform some action. In C, you'd express this as someAction(fred, someOptions), in C++, you would express this as fred.someAction(someOptions), and in Tk, you say:

```
.fred someAction someOptions
```

Note that the object name, .fred, starts with a dot.

As you'd expect, the legal values for someAction will depend on the widget's class: .fred disable works if fred is a button (fred gets greyed out), but does not work if fred is a label (disabling of labels is not supported in Tk).

The legal values of someOptions is action dependent. Some actions, like disable, require no arguments, others, like a text-entry box's delete command, would need arguments to specify what range of text to delete.

24.1.4 Mapping Basic Tk into Tkinter

Class commands in Tk correspond to class constructors in Tkinter.

```
button .fred              =====> fred = Button()
```

The master of an object is implicit in the new name given to it at creation time. In Tkinter, masters are specified explicitly.

```
button .panel.fred        =====> fred = Button(panel)
```

The configuration options in Tk are given in lists of hyphened tags followed by values. In Tkinter, options are specified as keyword-arguments in the instance constructor, and keyword-args for configure calls or as instance indices, in dictionary style, for established instances. See section Setting Options on setting options.

```
button .fred -fg red      =====> fred = Button(panel, fg = "red")
.fred configure -fg red   =====> fred["fg"] = red
                       OR ==> fred.config(fg = "red")
```

In Tk, to perform an action on a widget, use the widget name as a command, and follow it with an action name, possibly with arguments (options). In Tkinter, you call methods on the class instance to invoke actions on the widget. The actions (methods) that a given widget can perform are listed in the Tkinter.py module.

```
.fred invoke              =====> fred.invoke()
```

To give a widget to the packer (geometry manager), you call pack with optional arguments. In Tkinter, the Pack class holds all this functionality, and the various forms of the pack command are implemented as

methods. All widgets in Tkinter are subclassed from the Packer, and so inherit all the packing methods. See the Tix module documentation for additional information on the Form geometry manager.

```
pack .fred -side left      =====>  fred.pack(side = "left")
```

24.1.5 How Tk and Tkinter are Related

From the top down:

Your App Here (Python) A Python application makes a Tkinter call.

Tkinter (Python Module) This call (say, for example, creating a button widget), is implemented in the Tkinter module, which is written in Python. This Python function will parse the commands and the arguments and convert them into a form that makes them look as if they had come from a Tk script instead of a Python script.

tkinter (C) These commands and their arguments will be passed to a C function in the tkinter - note the lowercase - extension module.

Tk Widgets (C and Tcl) This C function is able to make calls into other C modules, including the C functions that make up the Tk library. Tk is implemented in C and some Tcl. The Tcl part of the Tk widgets is used to bind certain default behaviors to widgets, and is executed once at the point where the Python Tkinter module is imported. (The user never sees this stage).

Tk (C) The Tk part of the Tk Widgets implement the final mapping to ...

Xlib (C) the Xlib library to draw graphics on the screen.

24.1.6 Handy Reference

Setting Options

Options control things like the color and border width of a widget. Options can be set in three ways:

At object creation time, using keyword arguments

```
fred = Button(self, fg = "red", bg = "blue")
```

After object creation, treating the option name like a dictionary index

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

Use the config() method to update multiple attrs subsequent to object creation

```
fred.config(fg = "red", bg = "blue")
```

For a complete explanation of a given option and its behavior, see the Tk man pages for the widget in question.

Note that the man pages list "STANDARD OPTIONS" and "WIDGET SPECIFIC OPTIONS" for each widget. The former is a list of options that are common to many widgets, the latter are the options that are idiosyncratic to that particular widget. The Standard Options are documented on the options(3) man page.

No distinction between standard and widget-specific options is made in this document. Some options don't apply to some kinds of widgets. Whether a given widget responds to a particular option depends on the class of the widget; buttons have a command option, labels do not.

The options supported by a given widget are listed in that widget's man page, or can be queried at runtime by calling the config() method without arguments, or by calling the keys() method on that widget. The return value of these calls is a dictionary whose key is the name of the option as a string (for example, 'relief') and whose values are 5-tuples.

Some options, like bg are synonyms for common options with long names (bg is shorthand for "background"). Passing the config() method the name of a shorthand option will return a 2-tuple, not 5-tuple. The 2-tuple passed back will contain the name of the synonym and the "real" option (such as ('bg', 'background')).

Index	Meaning	Example
0	option name	'relief'
1	option name for database lookup	'relief'
2	option class for database lookup	'Relief'
3	default value	'raised'
4	current value	'groove'

Example:

```
>>> print fred.config()
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

Of course, the dictionary printed will include all the options available and their values. This is meant only as an example.

The Packer

The packer is one of Tk's geometry-management mechanisms. Geometry managers are used to specify the relative positioning of the positioning of widgets within their container - their mutual master. In contrast to the more cumbersome placer (which is used less commonly, and we do not cover here), the packer takes qualitative relationship specification - above, to the left of, filling, etc - and works everything out to determine the exact placement coordinates for you.

The size of any master widget is determined by the size of the "slave widgets" inside. The packer is used to control where slave widgets appear inside the master into which they are packed. You can pack widgets into frames, and frames into other frames, in order to achieve the kind of layout you desire. Additionally, the arrangement is dynamically adjusted to accommodate incremental changes to the configuration, once it is packed.

Note that widgets do not appear until they have had their geometry specified with a geometry manager. It's a common early mistake to leave out the geometry specification, and then be surprised when the widget is created but nothing appears. A widget will appear only after it has had, for example, the packer's pack() method applied to it.

The pack() method can be called with keyword-option/value pairs that control where the widget is to appear within its container, and how it is to behave when the main application window is resized. Here are some examples:

```
fred.pack()                     # defaults to side = "top"
fred.pack(side = "left")
fred.pack(expand = 1)
```

Packer Options

For more extensive information on the packer and the options that it can take, see the man pages and page 183 of John Ousterhout's book.

anchor Anchor type. Denotes where the packer is to place each slave in its parcel.

expand Boolean, 0 or 1.

fill Legal values: `'x'`, `'y'`, `'both'`, `'none'`.

ipadx and ipady A distance - designating internal padding on each side of the slave widget.

padx and pady A distance - designating external padding on each side of the slave widget.

side Legal values are: `'left'`, `'right'`, `'top'`, `'bottom'`.

Coupling Widget Variables

The current-value setting of some widgets (like text entry widgets) can be connected directly to application variables by using special options. These options are variable, textvariable, onvalue, offvalue, and value. This connection works both ways: if the variable changes for any reason, the widget it's connected to will be updated to reflect the new value.

Unfortunately, in the current implementation of Tkinter it is not possible to hand over an arbitrary Python variable to a widget through a variable or textvariable option. The only kinds of variables for which this works are variables that are subclassed from a class called Variable, defined in the Tkinter module.

There are many useful subclasses of Variable already defined: StringVar, IntVar, DoubleVar, and BooleanVar. To read the current value of such a variable, call the get() method on it, and to change its value you call the set() method. If you follow this protocol, the widget will always track the value of the variable, with no further intervention on your part.

For example:

```
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # here is the application variable
        self.contents = StringVar()
        # set it to some value
        self.contents.set("this is a variable")
        # tell the entry widget to watch this variable
        self.entrythingy["textvariable"] = self.contents

        # and here we get a callback when the user hits return.
        # we will have the program print out the value of the
        # application variable when the user hits return
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print "hi. contents of entry is now ---->", \
              self.contents.get()
```

The Window Manager

In Tk, there is a utility command, wm, for interacting with the window manager. Options to the wm command allow you to control things like titles, placement, icon bitmaps, and the like. In Tkinter, these commands have been implemented as methods on the Wm class. Toplevel widgets are subclassed from the Wm class, and so can call the Wm methods directly.

To get at the toplevel window that contains a given widget, you can often just refer to the widget's master. Of course if the widget has been packed inside of a frame, the master won't represent a toplevel window. To get at the toplevel window that contains an arbitrary widget, you can call the _root() method. This method begins with an underscore to denote the fact that this function is part of the implementation, and not an interface to Tk functionality.

Here are some examples of typical usage:

```
from Tkinter import *
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()


# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Tk Option Data Types

anchor Legal values are points of the compass: "n", "ne", "e", "se", "s", "sw", "w", "nw", and also "center".

bitmap There are eight built-in, named bitmaps: 'error', 'gray25', 'gray50', 'hourglass', 'info', 'questhead', 'question', 'warning'. To specify an X bitmap filename, give the full path to the file, preceded with an @, as in "@/usr/contrib/bitmap/gumby.bit".

boolean You can pass integers 0 or 1 or the strings "yes" or "no".

callback This is any Python function that takes no arguments. For example:

```
def print_it():
        print "hi there"
fred["command"] = print_it
```

color Colors can be given as the names of X colors in the rgb.txt file, or as strings representing RGB values in 4 bit: "#RGB", 8 bit: "#RRGGBB", 12 bit" "#RRRGGGBBB", or 16 bit "#RRRRGGGGBBBB" ranges, where R,G,B here represent any legal hex digit. See page 160 of Ousterhout's book for details.

cursor The standard X cursor names from cursorfont.h can be used, without the XC_ prefix. For example to get a hand cursor (XC_hand2), use the string "hand2". You can also specify a bitmap and mask file of your own. See page 179 of Ousterhout's book.

distance Screen distances can be specified in either pixels or absolute distances. Pixels are given as numbers and absolute distances as strings, with the trailing character denoting units: c for centimetres, i for inches, m for millimetres, p for printer's points. For example, 3.5 inches is expressed as "3.5i".

font Tk uses a list font name format, such as {courier 10 bold}. Font sizes with positive numbers are measured in points; sizes with negative numbers are measured in pixels.

geometry This is a string of the form widthxheight, where width and height are measured in pixels for most widgets (in characters for widgets displaying text). For example: fred["geometry"] = "200x100".

justify Legal values are the strings: "left", "center", "right", and "fill".

region This is a string with four space-delimited elements, each of which is a legal distance (see above). For example: "2 3 4 5" and "3i 2i 4.5i 2i" and "3c 2c 4c 10.43c" are all legal regions.

relief Determines what the border style of a widget will be. Legal values are: "raised", "sunken", "flat", "groove", and "ridge".

scrollcommand This is almost always the set() method of some scrollbar widget, but can be any widget method that takes a single argument. Refer to the file Demo/tkinter/matt/canvas-with-scrollbars.py in the Python source distribution for an example.

wrap: Must be one of: "none", "char", or "word".

Bindings and Events

The bind method from the widget command allows you to watch for certain events and to have a callback function trigger when that event type occurs. The form of the bind method is:

```
def bind(self, sequence, func, add='')
```

where:

sequence is a string that denotes the target kind of event. (See the bind man page and page 201 of John Ousterhout's book for details).

func is a Python function, taking one argument, to be invoked when the event occurs. An Event instance will be passed as the argument. (Functions deployed this way are commonly known as callbacks.)

add is optional, either '' or '+'. Passing an empty string denotes that this binding is to replace any other bindings that this event is associated with. Passing a '+' means that this function is to be added to the list of functions bound to this event type.

For example:

```
def turnRed(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turnRed)
```

Notice how the widget field of the event is being accessed in the turnRed() callback. This field contains the widget that caught the X event. The following table lists the other event fields you can access, and how they are denoted in Tk, which can be useful when referring to the Tk man pages.

Tk	Tkinter Event Field	Tk	Tkinter Event Field
%f	focus	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym

(continues on next page)

(continued from previous page)

%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

The index Parameter

A number of widgets require"index" parameters to be passed. These are used to point at a specific place in a Text widget, or to particular characters in an Entry widget, or to particular menu items in a Menu widget.

Entry widget indexes (index, view index, etc.) Entry widgets have options that refer to character positions in the text being displayed. You can use these Tkinter functions to access these special points in text widgets:

AtEnd() refers to the last position in the text

AtInsert() refers to the point where the text cursor is

AtSelFirst() indicates the beginning point of the selected text

AtSelLast() denotes the last point of the selected text and finally

At(x[, y]) refers to the character at pixel location x, y (with y not used in the case of a text entry widget, which contains a single line of text).

Text widget indexes The index notation for Text widgets is very rich and is best described in the Tk man pages.

Menu indexes (menu.invoke(), menu.entryconfig(), etc.) Some options and methods for menus manipulate specific menu entries. Anytime a menu index is needed for an option or a parameter, you may pass in:

- an integer which refers to the numeric position of the entry in the widget, counted from the top, starting with 0;
- the string `'active'`, which refers to the menu position that is currently under the cursor;
- the string "last" which refers to the last menu item;
- An integer preceded by @, as in @6, where the integer is interpreted as a y pixel coordinate in the menu's coordinate system;
- the string "none", which indicates no menu entry at all, most often used with menu.activate() to deactivate all entries, and finally,
- a text string that is pattern matched against the label of the menu entry, as scanned from the top of the menu to the bottom. Note that this index type is considered after all the others, which means that matches for menu items labelled last, active, or none may be interpreted as the above literals, instead.

Images

Images of different formats can be created through the corresponding subclass of Tkinter.Image:

- BitmapImage for images in XBM format.
- PhotoImage for images in PGM, PPM, GIF and PNG formats. The latter is supported starting with Tk 8.6.

Either type of image is created through either the file or the data option (other options are available as well).

The image object can then be used wherever an image option is supported by some widget (e.g. labels, buttons, menus). In these cases, Tk will not keep a reference to the image. When the last Python reference to the image object is deleted, the image data is deleted as well, and Tk will display an empty box wherever the image was used.

See also:

The Pillow package adds support for formats such as BMP, JPEG, TIFF, and WebP, among others.

24.1.7 File Handlers

Tk allows you to register and unregister a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. Only one handler may be registered per file descriptor. Example code:

```
import Tkinter
widget = Tkinter.Tk()
mask = Tkinter.READABLE | Tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

This feature is not available on Windows.

Since you don't know how many bytes are available for reading, you may not want to use the BufferedIOBase or TextIOBase read() or readline() methods, since these will insist on reading a predefined number of bytes. For sockets, the recv() or recvfrom() methods will work fine; for other files, use raw reads or os.read(file. fileno(), maxbytecount).

Widget.tk.createfilehandler(file, mask, func)
: Registers the file handler callback function func. The file argument may either be an object with a fileno() method (such as a file or socket object), or an integer file descriptor. The mask argument is an ORed combination of any of the three constants below. The callback is called as follows:

```
callback(file, mask)
```

Widget.tk.deletefilehandler(file)
: Unregisters a file handler.

Tkinter.READABLE
Tkinter.WRITABLE
Tkinter.EXCEPTION
: Constants used in the mask arguments.

24.2 ttk — Tk themed widgets

The ttk module provides access to the Tk themed widget set, which has been introduced in Tk 8.5. If Python is not compiled against Tk 8.5 code may still use this module as long as Tile is installed. However, some features provided by the new Tk, like anti-aliased font rendering under X11, window transparency (on X11 you will need a composition window manager) will be missing.

The basic idea of ttk is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.

See also:

Tk Widget Styling Support The document which brought up theming support for Tk

24.2.1 Using Ttk

To start using Ttk, import its module:

```
import ttk
```

But code like this:

```
from Tkinter import *
```

may optionally want to use this:

```
from Tkinter import *
from ttk import *
```

And then several ttk widgets (Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar) will automatically substitute for the Tk widgets.

This has the direct benefit of using the new widgets, giving better look & feel across platforms, but be aware that they are not totally compatible. The main difference is that widget options such as "fg", "bg" and others related to widget styling are no longer present in Ttk widgets. Use ttk.Style to achieve the same (or better) styling.

See also:

Converting existing applications to use the Tile widgets A text which talks in Tcl terms about differences typically found when converting applications to use the new widgets.

24.2.2 Ttk Widgets

Ttk comes with 17 widgets, 11 of which already exist in Tkinter: Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar. The 6 new widget classes are: Combobox, Notebook, Progressbar, Separator, Sizegrip and Treeview. All of these classes are subclasses of Widget.

As said previously, you will notice changes in look-and-feel as well in the styling code. To demonstrate the latter, a very simple example is shown below.

Tk code:

```
l1 = Tkinter.Label(text="Test", fg="black", bg="white")
l2 = Tkinter.Label(text="Test", fg="black", bg="white")
```

Corresponding Ttk code:

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

For more information about TtkStyling read the Style class documentation.

24.2.3 Widget

ttk.Widget defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

Standard Options

All the ttk widgets accept the following options:

Option	Description
class	Specifies the window class. The class is used when querying the option database for the window's other options, to determine the default bindtags for the window, and to select the widget's default layout and style. This is a read-only option which may only be specified when the window is created.
cursor	Specifies the mouse cursor to be used for the widget. If set to the empty string (the default), the cursor is inherited from the parent widget.
takefocus	Determines whether the window accepts the focus during keyboard traversal. 0, 1 or an empty string is returned. If 0, the window should be skipped entirely during keyboard traversal. If 1, the window should receive the input focus as long as it is viewable. An empty string means that the traversal scripts make the decision about whether or not to focus on the window.
style	May be used to specify a custom widget style.

Scrollable Widget Options

The following options are supported by widgets that are controlled by a scrollbar.

option	description
xscrollcommand	Used to communicate with horizontal scrollbars. When the view in the widget's window changes, the widget will generate a Tcl command based on the scrollcommand. Usually this option consists of the Scrollbar.set() method of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.
yscrollcommand	Used to communicate with vertical scrollbars. For more information, see above.

Label Options

The following options are supported by labels, buttons and other button-like widgets.

option	description
text	Specifies a text string to be displayed inside the widget.
textvariable	Specifies a name whose value will be used in place of the text option resource.
underline	If set, specifies the index (0-based) of a character to underline in the text string. The underline character is used for mnemonic activation.
image	Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list is a sequence of statespec/value pairs as defined by Style.map(), specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same size.
compound	Specifies how to display the image relative to the text, in the case both text and image options are present. Valid values are: • text: display text only • image: display image only • top, bottom, left, right: display image above, below, left of, or right of the text, respectively. • none: the default. display the image if present, otherwise the text.
width	If greater than zero, specifies how much space, in character widths, to allocate for the text label; if less than zero, specifies a minimum width. If zero or unspecified, the natural width of the text label is used.

Compatibility Options

option	description
state	May be set to "normal" or "disabled" to control the "disabled" state bit. This is a write-only option: setting it changes the widget state, but the Widget.state() method does not affect this option.

Widget States

The widget state is a bitmap of independent state flags.

flag	description
active	The mouse cursor is over the widget and pressing a mouse button will cause some action to occur.
disabled	Widget is disabled under program control.
focus	Widget has keyboard focus.
pressed	Widget is being pressed.
selected	"On", "true", or "current" for things like Checkbuttons and radiobuttons.
background	Windows and Mac have a notion of an "active" or foreground window. The background state is set for widgets in a background window, and cleared for those in the foreground window.
readonly	Widget should not allow user modification.
alternate	A widget-specific alternate display format.
invalid	The widget's value is invalid.

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is off.

ttk.Widget

Besides the methods described below, the ttk.Widget class supports the Tkinter.Widget.cget() and Tkinter.Widget.configure() methods.

class ttk.Widget

identify(x, y)
: Returns the name of the element at position x y, or the empty string if the point does not lie within any element.

  x and y are pixel coordinates relative to the widget.

instate(statespec, callback=None, *args, **kw)
: Test the widget's state. If a callback is not specified, returns True if the widget state matches statespec and False otherwise. If callback is specified then it is called with args if widget state matches statespec.

state([statespec=None])
: Modify or read widget state. If statespec is specified, sets the widget state accordingly and returns a new statespec indicating which flags were changed. If statespec is not specified, returns the currently-enabled state flags.

statespec will usually be a list or a tuple.

24.2.4 Combobox

The ttk.Combobox widget combines a text field with a pop-down list of values. This widget is a subclass of Entry.

Besides the methods inherited from Widget (Widget.cget(), Widget.configure(), Widget.identify(), Widget.instate() and Widget.state()) and those inherited from Entry (Entry.bbox(), Entry.delete(), Entry.icursor(), Entry.index(), Entry.insert(), Entry.selection(), Entry.xview()), this class has some other methods, described at ttk.Combobox.

Options

This widget accepts the following options:

option	description
exportselection	Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking Misc.selection_get(), for example).
justify	Specifies how the text is aligned within the widget. One of "left", "center", or "right".
height	Specifies the height of the pop-down listbox, in rows.
postcommand	A script (possibly registered with Misc.register()) that is called immediately before displaying the values. It may specify which values to display.
state	One of "normal", "readonly", or "disabled". In the "readonly" state, the value may not be edited directly, and the user can only select one of the values from the dropdown list. In the "normal" state, the text field is directly editable. In the "disabled" state, no interaction is possible.
textvariable	Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See Tkinter.StringVar.
values	Specifies the list of values to display in the drop-down listbox.
width	Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget's font.

Virtual events

The combobox widget generates a <<ComboboxSelected>> virtual event when the user selects an element from the list of values.

ttk.Combobox

class ttk.Combobox

current([newindex=None])
: If newindex is specified, sets the combobox value to the element position newindex. Otherwise, returns the index of the current value or -1 if the current value is not in the values list.

get()
: Returns the current value of the combobox.

set(value)
: Sets the value of the combobox to value.

24.2.5 Notebook

The Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently-displayed window.

Options

This widget accepts the following specific options:

option	description
height	If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.
padding	Specifies the amount of extra space to add around the outside of the notebook. The padding is a list of up to four length specifications: left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.
width	If present and greater than zero, specifies the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.

Tab Options

There are also specific options for tabs:

option	description
state	Either “normal”, “disabled” or “hidden”. If “disabled”, then the tab is not selectable. If “hidden”, then the tab is not shown.
sticky	Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters “n”, “s”, “e” or “w”. Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the grid() geometry manager.
padding	Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.
text	Specifies a text to be displayed in the tab.
image	Specifies an image to display in the tab. See the option image described in Widget.
compound	Specifies how to display the image relative to the text, in the case both text and image options are present. See Label Options for legal values.
underline	Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if Notebook.enable_traversal() is called.

Tab Identifiers

The tab_id present in several methods of ttk.Notebook may take any of the following forms:

- An integer between zero and the number of tabs.

- The name of a child window.
- A positional specification of the form "@x,y", which identifies the tab.
- The literal string "current", which identifies the currently-selected tab.
- The literal string "end", which returns the number of tabs (only valid for Notebook.index()).

Virtual Events

This widget generates a <<NotebookTabChanged>> virtual event after a new tab is selected.

ttk.Notebook

class ttk.Notebook

add(child, **kw)
: Adds a new tab to the notebook.

  If window is currently managed by the notebook but hidden, it is restored to its previous position.

  See Tab Options for the list of available options.

forget(tab_id)
: Removes the tab specified by tab_id, unmaps and unmanages the associated window.

hide(tab_id)
: Hides the tab specified by tab_id.

  The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the add() command.

identify(x, y)
: Returns the name of the tab element at position x, y, or the empty string if none.

index(tab_id)
: Returns the numeric index of the tab specified by tab_id, or the total number of tabs if tab_id is the string "end".

insert(pos, child, **kw)
: Inserts a pane at the specified position.

  pos is either the string "end", an integer index, or the name of a managed child. If child is already managed by the notebook, moves it to the specified position.

  See Tab Options for the list of available options.

select([tab_id])
: Selects the specified tab_id.

  The associated child window will be displayed, and the previously-selected window (if different) is unmapped. If tab_id is omitted, returns the widget name of the currently selected pane.

tab(tab_id, option=None, **kw)
: Query or modify the options of the specific tab_id.

  If kw is not given, returns a dictionary of the tab option values. If option is specified, returns the value of that option. Otherwise, sets the options to the corresponding values.

tabs()
: Returns a list of windows managed by the notebook.

enable_traversal()
Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows:

- Control-Tab: selects the tab following the currently selected one.
- Shift-Control-Tab: selects the tab preceding the currently selected one.
- Alt-K: where K is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.

24.2.6 Progressbar

The ttk.Progressbar widget shows the status of a long-running operation. It can operate in two modes: determinate mode shows the amount completed relative to the total amount of work to be done, and indeterminate mode provides an animated display to let the user know that something is happening.

Options

This widget accepts the following specific options:

option	description
orient	One of "horizontal" or "vertical". Specifies the orientation of the progress bar.
length	Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).
mode	One of "determinate" or "indeterminate".
maximum	A number specifying the maximum value. Defaults to 100.
value	The current value of the progress bar. In "determinate" mode, this represents the amount of work completed. In "indeterminate" mode, it is interpreted as modulo maximum; that is, the progress bar completes one "cycle" when its value increases by maximum.
variable	A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.
phase	Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the current theme to provide additional animation effects.

ttk.Progressbar

class ttk.Progressbar

start([interval])
Begin autoincrement mode: schedules a recurring timer event that calls Progressbar.step() every interval milliseconds. If omitted, interval defaults to 50 milliseconds.

step([amount])
Increments the progress bar's value by amount.

amount defaults to 1.0 if omitted.

stop()
Stop autoincrement mode: cancels any recurring timer event initiated by Progressbar.start() for this progress bar.

24.2.7 Separator

The ttk.Separator widget displays a horizontal or vertical separator bar.

It has no other methods besides the ones inherited from ttk.Widget.

Options

This widget accepts the following specific option:

option	description
orient	One of "horizontal" or "vertical". Specifies the orientation of the separator.

24.2.8 Sizegrip

The ttk.Sizegrip widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

This widget has neither specific options nor specific methods, besides the ones inherited from ttk.Widget.

Platform-specific notes

- On Mac OS X, toplevel windows automatically include a built-in size grip by default. Adding a Sizegrip is harmless, since the built-in grip will just mask the widget.

Bugs

- If the containing toplevel's position was specified relative to the right or bottom of the screen (e.g. ....), the Sizegrip widget will not resize the window.
- This widget supports only "southeast" resizing.

24.2.9 Treeview

The ttk.Treeview widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

The order in which data values are displayed may be controlled by setting the widget option displaycolumns. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option columns. See Column Identifiers.

Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named {}. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The Treeview widget supports horizontal and vertical scrolling, according to the options described in Scrollable Widget Options and the methods Treeview.xview() and Treeview.yview().

Options

This widget accepts the following specific options:

option	description
columns	A list of column identifiers, specifying the number of columns and their names.
displaycolumns	A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string "#all".
height	Specifies the number of rows which should be visible. Note: the requested width is determined from the sum of the column widths.
padding	Specifies the internal padding for the widget. The padding is a list of up to four length specifications.
selectmode	Controls how the built-in class bindings manage the selection. One of "extended", "browse" or "none". If set to "extended" (the default), multiple items may be selected. If "browse", only a single item will be selected at a time. If "none", the selection will not be changed. Note that the application code and tag bindings can set the selection however they wish, regardless of the value of this option.
show	A list containing zero or more of the following values, specifying which elements of the tree to display. • tree: display tree labels in column #0. • headings: display the heading row. The default is "tree headings", i.e., show all elements. Note: Column #0 always refers to the tree column, even if show="tree" is not specified.

Item Options

The following item options may be specified for items in the insert and item widget commands.

option	description
text	The textual label to display for the item.
image	A Tk Image, displayed to the left of the label.
values	The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.
open	True/False value indicating whether the item's children should be displayed or hidden.
tags	A list of tags associated with this item.

Tag Options

The following options may be specified on tags:

option	description
foreground	Specifies the text foreground color.
background	Specifies the cell or item background color.
font	Specifies the font to use when drawing text.
image	Specifies the item image, in case the item's image option is empty.

Column Identifiers

Column identifiers take any of the following forms:

- A symbolic name from the list of columns option.
- An integer n, specifying the nth data column.
- A string of the form #n, where n is an integer, specifying the nth display column.

Notes:

- Item's option values may be displayed in a different order than the order in which they are stored.
- Column #0 always refers to the tree column, even if show="tree" is not specified.

A data column number is an index into an item's option values list; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, column #0 always refers to the tree column.

Virtual Events

The Treeview widget generates the following virtual events.

event	description
<<TreeviewSelect>>	Generated whenever the selection changes.
<<TreeviewOpen>>	Generated just before settings the focus item to open=True.
<<TreeviewClose>>	Generated just after setting the focus item to open=False.

The Treeview.focus() and Treeview.selection() methods can be used to determine the affected item or items.

ttk.Treeview

class ttk.Treeview

bbox(item, column=None)
: Returns the bounding box (relative to the treeview widget's window) of the specified item in the form (x, y, width, height).

  If column is specified, returns the bounding box of that cell. If the item is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children([item])
Returns the list of children belonging to item.

If item is not specified, returns root children.

set_children(item, *newchildren)
Replaces item's child with newchildren.

Children present in item that are not present in newchildren are detached from the tree. No items in newchildren may be an ancestor of item. Note that not specifying newchildren results in detaching item's children.

column(column, option=None, **kw)
Query or modify the options for the specified column.

If kw is not given, returns a dict of the column option values. If option is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- id Returns the column name. This is a read-only option.
- anchor: One of the standard Tk anchor values. Specifies how the text in this column should be aligned with respect to the cell.
- minwidth: width The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.
- stretch: True/False Specifies whether the column's width should be adjusted when the widget is resized.
- width: width The width of the column in pixels.

To configure the tree column, call this with column = "#0"

delete(*items)
Delete all specified items and all their descendants.

The root item may not be deleted.

detach(*items)
Unlinks all of the specified items from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

exists(item)
Returns True if the specified item is present in the tree.

focus([item=None])
If item is specified, sets the focus item to item. Otherwise, returns the current focus item, or '' if there is none.

heading(column, option=None, **kw)
Query or modify the heading options for the specified column.

If kw is not given, returns a dict of the heading option values. If option is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- text: text The text to display in the column heading.

- image: imageName Specifies an image to display to the right of the column heading.
- anchor: anchor Specifies how the heading text should be aligned. One of the standard Tk anchor values.
- command: callback A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with column = "#0".

identify(component, x, y)
Returns a description of the specified component under the point given by x and y, or the empty string if no such component is present at that position.

identify_row(y)
Returns the item ID of the item at position y.

identify_column(x)
Returns the data column identifier of the cell at position x.

The tree column has ID #0.

identify_region(x, y)
Returns one of:

region	meaning
heading	Tree heading area.
separator	Space between two columns headings.
tree	The tree area.
cell	A data cell.

Availability: Tk 8.6.

identify_element(x, y)
Returns the element at position x, y.

Availability: Tk 8.6.

index(item)
Returns the integer index of item within its parent's list of children.

insert(parent, index, iid=None, **kw)
Creates a new item and returns the item identifier of the newly created item.

parent is the item ID of the parent item, or the empty string to create a new top-level item. index is an integer, or the value "end", specifying where in the list of parent's children to insert the new item. If index is less than or equal to zero, the new node is inserted at the beginning; if index is greater than or equal to the current number of children, it is inserted at the end. If iid is specified, it is used as the item identifier; iid must not already exist in the tree. Otherwise, a new unique identifier is generated.

See Item Options for the list of available points.

item(item[, option[, **kw]])
Query or modify the options for the specified item.

If no options are given, a dict with options/values for the item is returned. If option is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by kw.

move(item, parent, index)
Moves item to position index in parent's list of children.

It is illegal to move an item under one of its descendants. If index is less than or equal to zero, item is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If item was detached it is reattached.

next(item)
Returns the identifier of item's next sibling, or '' if item is the last child of its parent.

parent(item)
Returns the ID of the parent of item, or '' if item is at the top level of the hierarchy.

prev(item)
Returns the identifier of item's previous sibling, or '' if item is the first child of its parent.

reattach(item, parent, index)
An alias for Treeview.move().

see(item)
Ensure that item is visible.

Sets all of item's ancestors open option to True, and scrolls the widget if necessary so that item is within the visible portion of the tree.

selection([selop=None[, items=None]])
If selop is not specified, returns selected items. Otherwise, it will act according to the following selection methods.

selection_set(items)
items becomes the new selection.

selection_add(items)
Add items to the selection.

selection_remove(items)
Remove items from the selection.

selection_toggle(items)
Toggle the selection state of each item in items.

set(item, column=None, value=None)
With one argument, returns a dictionary of column/value pairs for the specified item. With two arguments, returns the current value of the specified column. With three arguments, sets the value of given column in given item to the specified value.

tag_bind(tagname, sequence=None, callback=None)
Bind a callback for the given event sequence to the tag tagname. When an event is delivered to an item, the callbacks for each of the item's tags option are called.

tag_configure(tagname, option=None, **kw)
Query or modify the options for the specified tagname.

If kw is not given, returns a dict of the option settings for tagname. If option is specified, returns the value for that option for the specified tagname. Otherwise, sets the options to the corresponding values for the given tagname.

tag_has(tagname[, item])
If item is specified, returns 1 or 0 depending on whether the specified item has the given tagname. Otherwise, returns a list of all items that have the specified tag.

Availability: Tk 8.6

xview(*args)
Query or modify horizontal position of the treeview.

yview(*args)
Query or modify vertical position of the treeview.

24.2.10 Ttk Styling

Each widget in ttk is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget's class name, but it may be overridden by the widget's style option. If the class name of a widget is unknown, use the method Misc.winfo_class() (somewidget.winfo_class()).

See also:

Tcl'2004 conference presentation This document explains how the theme engine works

class ttk.Style
This class is used to manipulate the style database.

configure(style, query_opt=None, **kw)
Query or set the default value of the specified option(s) in style.

Each key in kw is an option and each value is a string identifying the value for that option.

For example, to change every default button to be a flat button with some padding and a different background color do:

```
import ttk
import Tkinter

root = Tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map(style, query_opt=None, **kw)
Query or sets dynamic values of the specified option(s) in style.

Each key in kw is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or something else of your preference. A statespec is a compound of one or more states and then a value.

An example:

```
import Tkinter
import ttk

root = Tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()
```

(continues on next page)

(continued from previous page)

```
root.mainloop()
```

Note that the order of the (states, value) sequences for an option matters. In the previous example, if you change the order to [('active', 'blue'), ('pressed', 'red')] in the foreground option, for example, you would get a blue foreground when the widget is in the active or pressed states.

lookup(style, option, state=None, default=None)
Returns the value specified for option in style.

If state is specified, it is expected to be a sequence of one or more states. If the default argument is set, it is used as a fallback value in case no specification for option is found.

To check what font a Button uses by default, do:

```
import ttk

print ttk.Style().lookup("TButton", "font")
```

layout(style, layoutspec=None)
Define the widget layout for given style. If layoutspec is omitted, return the layout specification for given style.

layoutspec, if specified, is expected to be a list or some other sequence type (excluding strings), where each item should be a tuple and the first item is the layout name and the second item should have the format described in Layouts.

To understand the format, see the following example (it is not intended to do anything useful):

```
import ttk
import Tkinter

root = Tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
  ("Menubutton.background", None),
  ("Menubutton.button", {"children":
      [("Menubutton.focus", {"children":
          [("Menubutton.padding", {"children":
              [("Menubutton.label", {"side": "left", "expand": 1})]
          })]
      })]
  }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create(elementname, etype, *args, **kw)
Create a new element in the current theme, of the given etype which is expected to be either “image”, “from” or “vsapi”. The latter is only available in Tk 8.6a for Windows XP and Vista and is not described here.

If “image” is used, args should contain the default image name followed by statespec/value pairs (this is the imagespec), and kw may have the following options:

- border=padding padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.
- height=height Specifies a minimum height for the element. If less than zero, the base image's height is used as a default.
- padding=padding Specifies the element's interior padding. Defaults to border's value if not specified.
- sticky=spec Specifies how the image is placed within the final parcel. spec contains zero or more characters "n", "s", "w", or "e".
- width=width Specifies a minimum width for the element. If less than zero, the base image's width is used as a default.

If "from" is used as the value of etype, element_create() will clone an existing element. args is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. kw is discarded.

element_names()
Returns the list of elements defined in the current theme.

element_options(elementname)
Returns the list of elementname's options.

theme_create(themename, parent=None, settings=None)
Create a new theme.

It is an error if themename already exists. If parent is specified, the new theme will inherit styles, elements and layouts from the parent theme. If settings are present they are expected to have the same syntax used for theme_settings().

theme_settings(themename, settings)
Temporarily sets the current theme to themename, apply specified settings and then restore the previous theme.

Each key in settings is a style and each value may contain the keys 'configure', 'map', 'layout' and 'element create' and they are expected to have the same format as specified by the methods Style.configure(), Style.map(), Style.layout() and Style.element_create() respectively.

As an example, let's change the Combobox for the default theme a bit:

```
import ttk
import Tkinter

root = Tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
  "TCombobox": {
      "configure": {"padding": 5},
      "map": {
          "background": [("active", "green2"),
                         ("!disabled", "green4")],
          "fieldbackground": [("!disabled", "green3")],
          "foreground": [("focus", "OliveDrab1"),
                         ("!disabled", "OliveDrab2")]
      }
  }
})
```

(continues on next page)

(continued from previous page)

```
combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names()
: Returns a list of all known themes.

theme_use([themename])
: If themename is not given, returns the theme in use. Otherwise, sets the current theme to themename, refreshes all widgets and emits a <<ThemeChanged>> event.

Layouts

A layout can be just None, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager: given an initial cavity, each element is allocated a parcel. Valid options/values are:

- side: whichside Specifies which side of the cavity to place the element; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.
- sticky: nswe Specifies where the element is placed inside its allocated parcel.
- unit: 0 or 1 If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of Widget.identify() et al. It's used for things like scrollbar thumbs with grips.
- children: [sublayout...] Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a Layout.

24.3 Tix — Extension widgets for Tk

The Tix (Tk Interface Extension) module provides an additional rich set of widgets. Although the standard Tk library has many useful widgets, they are far from complete. The Tix library provides most of the commonly needed widgets that are missing from standard Tk: HList, ComboBox, Control (a.k.a. SpinBox) and an assortment of scrollable widgets. Tix also includes many more widgets that are generally useful in a wide range of applications: NoteBook, FileEntry, PanedWindow, etc; there are more than 40 of them.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

Note: Tix has been renamed to tkinter.tix in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

See also:

Tix Homepage The home page for Tix. This includes links to additional documentation and downloads.

Tix Man Pages On-line version of the man pages and reference material.

Tix Programming Guide On-line version of the programmer's reference material.

Tix Development Applications Tix applications for development of Tix and Tkinter programs. Tide applications work under Tk or Tkinter, and include TixInspect, an inspector to remotely modify and debug Tix/Tk/Tkinter applications.

24.3.1 Using Tix

class Tix.Tix(screenName[, baseName[, className]])
Toplevel widget of Tix which represents mostly the main window of an application. It has an associated Tcl interpreter.

Classes in the Tix module subclasses the classes in the Tkinter module. The former imports the latter, so to use Tix with Tkinter, all you need to do is to import one module. In general, you can just import Tix, and replace the toplevel call to Tkinter.Tk with Tix.Tk:

```
import Tix
from Tkconstants import *
root = Tix.Tk()
```

To use Tix, you must have the Tix widgets installed, usually alongside your installation of the Tk widgets. To test your installation, try the following:

```
import Tix
root = Tix.Tk()
root.tk.eval('package require Tix')
```

If this fails, you have a Tk installation problem which must be resolved before proceeding. Use the environment variable TIX_LIBRARY to point to the installed Tix library directory, and make sure you have the dynamic object library (tix8183.dll or libtix8183.so) in the same directory that contains your Tk dynamic object library (tk8183.dll or libtk8183.so). The directory with the dynamic object library should also have a file called pkgIndex.tcl (case sensitive), which contains the line:

```
package ifneeded Tix 8.1 [list load "[file join $dir tix8183.dll]" Tix]
```

24.3.2 Tix Widgets

Tix introduces over 40 widget classes to the Tkinter repertoire. There is a demo of all the Tix widgets in the Demo/tix directory of the standard distribution.

Basic Widgets

class Tix.Balloon
A Balloon that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

class Tix.ButtonBox
The ButtonBox widget creates a box of buttons, such as is commonly used for Ok Cancel.

class Tix.ComboBox
The ComboBox widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwidget or selecting from the listbox subwidget.

class Tix.Control
The Control widget is also known as the SpinBox widget. The user can adjust the value by pressing the two arrow buttons or by entering the value directly into the entry. The new value will be checked against the user-defined upper and lower limits.

class Tix.LabelEntry
: The LabelEntry widget packages an entry widget and a label into one mega widget. It can be used to simplify the creation of "entry-form" type of interface.

class Tix.LabelFrame
: The LabelFrame widget packages a frame widget and a label into one mega widget. To create widgets inside a LabelFrame widget, one creates the new widgets relative to the frame subwidget and manage them inside the frame subwidget.

class Tix.Meter
: The Meter widget can be used to show the progress of a background job which may take a long time to execute.

class Tix.OptionMenu
: The OptionMenu creates a menu button of options.

class Tix.PopupMenu
: The PopupMenu widget can be used as a replacement of the tk_popup command. The advantage of the Tix PopupMenu widget is it requires less application code to manipulate.

class Tix.Select
: The Select widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

class Tix.StdButtonBox
: The StdButtonBox widget is a group of standard buttons for Motif-like dialog boxes.

File Selectors

class Tix.DirList
: The DirList widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class Tix.DirTree
: The DirTree widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class Tix.DirSelectDialog
: The DirSelectDialog widget presents the directories in the file system in a dialog window. The user can use this dialog window to navigate through the file system to select the desired directory.

class Tix.DirSelectBox
: The DirSelectBox is similar to the standard Motif(TM) directory-selection box. It is generally used for the user to choose a directory. DirSelectBox stores the directories mostly recently selected into a ComboBox widget so that they can be quickly selected again.

class Tix.ExFileSelectBox
: The ExFileSelectBox widget is usually embedded in a tixExFileSelectDialog widget. It provides a convenient method for the user to select files. The style of the ExFileSelectBox widget is very similar to the standard file dialog on MS Windows 3.1.

class Tix.FileSelectBox
: The FileSelectBox is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. FileSelectBox stores the files mostly recently selected into a ComboBox widget so that they can be quickly selected again.

class Tix.FileEntry
: The FileEntry widget can be used to input a filename. The user can type in the filename manually.

Alternatively, the user can press the button widget that sits next to the entry, which will bring up a file selection dialog.

Hierarchical ListBox

class Tix.HList
: The HList widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierarchy.

class Tix.CheckList
: The CheckList widget displays a list of items to be selected by the user. CheckList acts similarly to the Tk checkbutton or radiobutton widgets, except it is capable of handling many more items than checkbuttons or radiobuttons.

class Tix.Tree
: The Tree widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

Tabular ListBox

class Tix.TList
: The TList widget can be used to display data in a tabular format. The list entries of a TList widget are similar to the entries in the Tk listbox widget. The main differences are (1) the TList widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.

Manager Widgets

class Tix.PanedWindow
: The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally. The user changes the sizes of the panes by dragging the resize handle between two panes.

class Tix.ListNoteBook
: The ListNoteBook widget is very similar to the TixNoteBook widget: it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the hlist subwidget.

class Tix.NoteBook
: The NoteBook widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual “tabs” at the top of the NoteBook widget.

Image Types

The Tix module adds:

- pixmap capabilities to all Tix and Tkinter widgets to create color images from XPM files.

- Compound image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. For example, a compound image can be used to display a bitmap and a text string simultaneously in a Tk Button widget.

Miscellaneous Widgets

class Tix.InputOnly
The InputOnly widgets are to accept inputs from the user, which can be done with the bind command (Unix only).

Form Geometry Manager

In addition, Tix augments Tkinter by providing:

class Tix.Form
The Form geometry manager based on attachment rules for all Tk widgets.

24.3.3 Tix Commands

class Tix.tixCommand
The tix commands provide access to miscellaneous elements of Tix's internal state and the Tix application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

To view the current settings, the common usage is:

```
import Tix
root = Tix.Tk()
print root.tix_configure()
```

tixCommand.tix_configure(cnf=None **kw)
Query or modify the configuration options of the Tix application context. If no option is specified, returns a dictionary all of the available options. If option is specified with no value, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. Option may be any of the configuration options.

tixCommand.tix_cget(option)
Returns the current value of the configuration option given by option. Option may be any of the configuration options.

tixCommand.tix_getbitmap(name)
Locates a bitmap file of the name name.xpm or name in one of the bitmap directories (see the tix_addbitmapdir() method). By using tix_getbitmap(), you can avoid hard coding the pathnames of the bitmap files in your application. When successful, it returns the complete pathname of the bitmap file, prefixed with the character @. The returned value can be used to configure the bitmap option of the Tk and Tix widgets.

tixCommand.tix_addbitmapdir(directory)
Tix maintains a list of directories under which the tix_getimage() and tix_getbitmap() methods will search for image files. The standard bitmap directory is $TIX_LIBRARY/bitmaps. The tix_addbitmapdir() method adds directory into this list. By using this method, the image files of an applications can also be located using the tix_getimage() or tix_getbitmap() method.

tixCommand.tix_filedialog([dlgclass])
: Returns the file selection dialog that may be shared among different calls from this application. This method will create a file selection dialog widget when it is called the first time. This dialog will be returned by all subsequent calls to tix_filedialog(). An optional dlgclass parameter can be passed as a string to specified what type of file selection dialog widget is desired. Possible options are tix, FileSelectDialog or tixExFileSelectDialog.

tixCommand.tix_getimage(self, name)
: Locates an image file of the name name.xpm, name.xbm or name.ppm in one of the bitmap directories (see the tix_addbitmapdir() method above). If more than one file with the same name (but different extensions) exist, then the image type is chosen according to the depth of the X display: xbm images are chosen on monochrome displays and color images are chosen on color displays. By using tix_getimage(), you can avoid hard coding the pathnames of the image files in your application. When successful, this method returns the name of the newly created image, which can be used to configure the image option of the Tk and Tix widgets.

tixCommand.tix_option_get(name)
: Gets the options maintained by the Tix scheme mechanism.

tixCommand.tix_resetoptions(newScheme, newFontSet[, newScmPrio])
: Resets the scheme and fontset of the Tix application to newScheme and newFontSet, respectively. This affects only those widgets created after this call. Therefore, it is best to call the resetoptions method before the creation of any widgets in a Tix application.

  The optional parameter newScmPrio can be given to reset the priority level of the Tk options set by the Tix schemes.

  Because of the way Tk handles the X option database, after Tix has been has imported and inited, it is not possible to reset the color schemes and font sets using the tix_config() method. Instead, the tix_resetoptions() method must be used.

24.4 ScrolledText — Scrolled Text Widget

The ScrolledText module provides a class of the same name which implements a basic text widget which has a vertical scroll bar configured to do the "right thing." Using the ScrolledText class is a lot easier than setting up a text widget and scroll bar directly. The constructor is the same as that of the Tkinter.Text class.

Note: ScrolledText has been renamed to tkinter.scrolledtext in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

The text widget and scrollbar are packed together in a Frame, and the methods of the Grid and Pack geometry managers are acquired from the Frame object. This allows the ScrolledText widget to be used directly to achieve most normal geometry management behavior.

Should more specific control be necessary, the following attributes are available:

ScrolledText.frame
: The frame which surrounds the text and scroll bar widgets.

ScrolledText.vbar
: The scroll bar widget.

24.5 turtle — Turtle graphics for Tk

24.5.1 Introduction

Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzig and Seymour Papert in 1966.

Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an import turtle, give it the command turtle.forward(15), and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command turtle.right(25), and it rotates in-place 25 degrees clockwise.

By combining together these and similar commands, intricate shapes and pictures can easily be drawn.

The turtle module is an extended reimplementation of the same-named module from the Python standard distribution up to version Python 2.5.

It tries to keep the merits of the old turtle module and to be (nearly) 100% compatible with it. This means in the first place to enable the learning programmer to use all the commands, classes and methods interactively when using the module from within IDLE run with the -n switch.

The turtle module provides turtle graphics primitives, in both object-oriented and procedure-oriented ways. Because it uses Tkinter for the underlying graphics, it needs a version of Python installed with Tk support.

The object-oriented interface uses essentially two+two classes:

1. The TurtleScreen class defines graphics windows as a playground for the drawing turtles. Its constructor needs a Tkinter.Canvas or a ScrolledCanvas as argument. It should be used when turtle is used as part of some application.

   The function Screen() returns a singleton object of a TurtleScreen subclass. This function should be used when turtle is used as a standalone tool for doing graphics. As a singleton object, inheriting from its class is not possible.

   All methods of TurtleScreen/Screen also exist as functions, i.e. as part of the procedure-oriented interface.

2. RawTurtle (alias: RawPen) defines Turtle objects which draw on a TurtleScreen. Its constructor needs a Canvas, ScrolledCanvas or TurtleScreen as argument, so the RawTurtle objects know where to draw.

   Derived from RawTurtle is the subclass Turtle (alias: Pen), which draws on "the" Screen - instance which is automatically created, if not already present.

   All methods of RawTurtle/Turtle also exist as functions, i.e. part of the procedure-oriented interface.

The procedural interface provides functions which are derived from the methods of the classes Screen and Turtle. They have the same names as the corresponding methods. A screen object is automatically created whenever a function derived from a Screen method is called. An (unnamed) turtle object is automatically created whenever any of the functions derived from a Turtle method is called.

To use multiple turtles an a screen one has to use the object-oriented interface.

Note: In the following documentation the argument list for functions is given. Methods, of course, have the additional first argument self which is omitted here.

24.5.2 Overview over available Turtle and Screen methods

Turtle methods

Turtle motion

Move and draw

forward() | fd()
backward() | bk() | back()
right() | rt()
left() | lt()
goto() | setpos() | setposition()
setx()
sety()
setheading() | seth()
home()
circle()
dot()
stamp()
clearstamp()
clearstamps()
undo()
speed()

Tell Turtle's state

position() | pos()
towards()
xcor()
ycor()
heading()
distance()

Setting and measurement

degrees()
radians()

Pen control

Drawing state

pendown() | pd() | down()
penup() | pu() | up()
pensize() | width()
pen()
isdown()

Color control

color()
pencolor()
fillcolor()

Filling

fill()

begin_fill()
end_fill()

More drawing control

reset()
clear()
write()

Turtle state

Visibility

showturtle() | st()
hideturtle() | ht()
isvisible()

Appearance

shape()
resizemode()
shapesize() | turtlesize()
settiltangle()
tiltangle()
tilt()

Using events

onclick()
onrelease()
ondrag()
mainloop() | done()

Special Turtle methods

begin_poly()
end_poly()
get_poly()
clone()
getturtle() | getpen()
getscreen()
setundobuffer()
undobufferentries()
tracer()
window_width()
window_height()

Methods of TurtleScreen/Screen

Window control

bgcolor()
bgpic()
clear() | clearscreen()
reset() | resetscreen()

screensize()
setworldcoordinates()

Animation control

delay()
tracer()
update()

Using screen events

listen()
onkey()
onclick() | onscreenclick()
ontimer()

Settings and special methods

mode()
colormode()
getcanvas()
getshapes()
register_shape() | addshape()
turtles()
window_height()
window_width()

Methods specific to Screen

bye()
exitonclick()
setup()
title()

24.5.3 Methods of RawTurtle/Turtle and corresponding functions

Most of the examples in this section refer to a Turtle instance called turtle.

Turtle motion

turtle.forward(distance)
turtle.fd(distance)

Parameters distance – a number (integer or float)

Move the turtle forward by the specified distance, in the direction the turtle is headed.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

turtle.back(distance)
turtle.bk(distance)
turtle.backward(distance)

Parameters distance – a number

Move the turtle backward by distance, opposite to the direction the turtle is headed. Do not change the turtle's heading.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

turtle.right(angle)
turtle.rt(angle)

Parameters angle – a number (integer or float)

Turn turtle right by angle units. (Units are by default degrees, but can be set via the degrees() and radians() functions.) Angle orientation depends on the turtle mode, see mode().

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

turtle.left(angle)
turtle.lt(angle)

Parameters angle – a number (integer or float)

Turn turtle left by angle units. (Units are by default degrees, but can be set via the degrees() and radians() functions.) Angle orientation depends on the turtle mode, see mode().

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.goto(x, y=None)
turtle.setpos(x, y=None)
turtle.setposition(x, y=None)

Parameters

- x – a number or a pair/vector of numbers
- y – a number or None

If y is None, x must be a pair of coordinates or a Vec2D (e.g. as returned by pos()).

Move turtle to an absolute position. If the pen is down, draw line. Do not change the turtle's orientation.

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
```

(continues on next page)

(continued from previous page)

```
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

turtle.setx(x)

Parameters x – a number (integer or float)

Set the turtle's first coordinate to x, leave second coordinate unchanged.

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.sety(y)

Parameters y – a number (integer or float)

Set the turtle's second coordinate to y, leave first coordinate unchanged.

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.setheading(to_angle)
turtle.seth(to_angle)

Parameters to_angle – a number (integer or float)

Set the orientation of the turtle to to_angle. Here are some common directions in degrees:

standard mode	logo mode
0 - east	0 - north
90 - north	90 - east
180 - west	180 - south
270 - south	270 - west

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.home()

Move turtle to the origin – coordinates (0,0) – and set its heading to its start-orientation (which depends on the mode, see mode()).

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.circle(radius, extent=None, steps=None)

Parameters

- radius – a number
- extent – a number (or None)
- steps – an integer (or None)

Draw a circle with given radius. The center is radius units left of the turtle; extent – an angle – determines which part of the circle is drawn. If extent is not given, draw the entire circle. If extent is not a full circle, one endpoint of the arc is the current pen position. Draw the arc in counterclockwise direction if radius is positive, otherwise in clockwise direction. Finally the direction of the turtle is changed by the amount of extent.

As the circle is approximated by an inscribed regular polygon, steps determines the number of steps to use. If not given, it will be calculated automatically. May be used to draw regular polygons.

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

turtle.dot(size=None, *color)

Parameters

- size – an integer >= 1 (if given)
- color – a colorstring or a numeric color tuple

Draw a circular dot with diameter size, using color. If size is not given, the maximum of pensize+4 and 2*pensize is used.

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
```

(continues on next page)

(continued from previous page)

```
(100.00,-0.00)
>>> turtle.heading()
0.0
```

turtle.stamp()

Stamp a copy of the turtle shape onto the canvas at the current turtle position. Return a stamp_id for that stamp, which can be used to delete it by calling clearstamp(stamp_id).

```
>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)
```

turtle.clearstamp(stampid)

Parameters stampid – an integer, must be return value of previous stamp() call

Delete stamp with given stampid.

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

turtle.clearstamps(n=None)

Parameters n – an integer (or None)

Delete all or first/last n of turtle's stamps. If n is None, delete all stamps, if n > 0 delete first n stamps, else if n < 0 delete last n stamps.

```
>>> for i in range(8):
...     turtle.stamp(); turtle.fd(30)
13
14
15
16
17
18
19
20
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.undo()

Undo (repeatedly) the last turtle action(s). Number of available undo actions is determined by the size of the undobuffer.

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
```

(continues on next page)

(continued from previous page)

```
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.speed(speed=None)

Parameters speed – an integer in the range 0..10 or a speedstring (see below)

Set the turtle's speed to an integer value in the range 0..10. If no argument is given, return current speed.

If input is a number greater than 10 or smaller than 0.5, speed is set to 0. Speedstrings are mapped to speedvalues as follows:

- "fastest": 0
- "fast": 10
- "normal": 6
- "slow": 3
- "slowest": 1

Speeds from 1 to 10 enforce increasingly faster animation of line drawing and turtle turning.

Attention: speed = 0 means that no animation takes place. forward/back makes turtle jump and likewise left/right make the turtle turn instantly.

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

Tell Turtle's state

turtle.position()
turtle.pos()

Return the turtle's current location (x,y) (as a Vec2D vector).

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.towards(x, y=None)

Parameters

- x – a number or a pair/vector of numbers or a turtle instance
- y – a number if x is a number, else None

Return the angle between the line from turtle position to position specified by (x,y), the vector or the other turtle. This depends on the turtle's start orientation which depends on the mode - "standard"/"world" or "logo").

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.xcor()

Return the turtle's x coordinate.

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print turtle.xcor()
64.2787609687
```

turtle.ycor()

Return the turtle's y coordinate.

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print turtle.pos()
(50.00,86.60)
>>> print turtle.ycor()
86.6025403784
```

turtle.heading()

Return the turtle's current heading (value depends on the turtle mode, see mode()).

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.distance(x, y=None)

Parameters

- x – a number or a pair/vector of numbers or a turtle instance
- y – a number if x is a number, else None

Return the distance from the turtle to (x,y), the given vector, or the given other turtle, in turtle step units.

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

Settings for measurement

turtle.degrees(fullcircle=360.0)

Parameters fullcircle – a number

Set angle measurement units, i.e. set number of "degrees" for a full circle. Default value is 360 degrees.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon,
grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.radians()

Set the angle measurement units to radians. Equivalent to degrees(2*math.pi).

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

Pen control

Drawing state

turtle.pendown()
turtle.pd()
turtle.down()

Pull the pen down – drawing when moving.

turtle.penup()
turtle.pu()
turtle.up()

Pull the pen up – no drawing when moving.

turtle.pensize(width=None)
turtle.width(width=None)

Parameters width – a positive number

Set the line thickness to width or return it. If resizemode is set to "auto" and turtleshape is a polygon, that polygon is drawn with the same line thickness. If no argument is given, the current pensize is returned.

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)   # from here on lines of width 10 are drawn
```

turtle.pen(pen=None, **pendict)

Parameters

- pen – a dictionary with some or all of the below listed keys
- pendict – one or more keyword-arguments with the below listed keys as keywords

Return or set the pen's attributes in a "pen-dictionary" with the following key/value pairs:

- "shown": True/False
- "pendown": True/False
- "pencolor": color-string or color-tuple
- "fillcolor": color-string or color-tuple
- "pensize": positive number
- "speed": number in range 0..10
- "resizemode": "auto" or "user" or "noresize"
- "stretchfactor": (positive number, positive number)
- "outline": positive number
- "tilt": number

This dictionary can be used as argument for a subsequent call to pen() to restore the former pen-state. Moreover one or more of these attributes can be provided as keyword-arguments. This can be used to set several pen attributes in one statement.

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow'),
 ('pendown', False), ('pensize', 10), ('resizemode', 'noresize'),
 ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]
```

turtle.isdown()

Return True if pen is down, False if it's up.

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

Color control

turtle.pencolor(*args)

Return or set the pencolor.

Four input formats are allowed:

pencolor() Return the current pencolor as color specification string or as a tuple (see example). May be used as input to another color/pencolor/fillcolor call.

pencolor(colorstring) Set pencolor to colorstring, which is a Tk color specification string, such as "red", "yellow", or "#33cc8c".

pencolor((r, g, b)) Set pencolor to the RGB color represented by the tuple of r, g, and b. Each of r, g, and b must be in the range 0..colormode, where colormode is either 1.0 or 255 (see colormode()).

pencolor(r, g, b)

Set pencolor to the RGB color represented by r, g, and b. Each of r, g, and b must be in the range 0..colormode.

If turtleshape is a polygon, the outline of that polygon is drawn with the newly set pencolor.

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51, 204, 140)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50, 193, 143)
```

turtle.fillcolor(*args)

Return or set the fillcolor.

Four input formats are allowed:

fillcolor() Return the current fillcolor as color specification string, possibly in tuple format (see example). May be used as input to another color/pencolor/fillcolor call.

fillcolor(colorstring) Set fillcolor to colorstring, which is a Tk color specification string, such as "red", "yellow", or "#33cc8c".

fillcolor((r, g, b)) Set fillcolor to the RGB color represented by the tuple of r, g, and b. Each of r, g, and b must be in the range 0..colormode, where colormode is either 1.0 or 255 (see colormode()).

fillcolor(r, g, b)

Set fillcolor to the RGB color represented by r, g, and b. Each of r, g, and b must be in the range 0..colormode.

If turtleshape is a polygon, the interior of that polygon is drawn with the newly set fillcolor.

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> col = turtle.pencolor()
>>> col
(50, 193, 143)
>>> turtle.fillcolor(col)
>>> turtle.fillcolor()
(50, 193, 143)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255, 255, 255)
```

turtle.color(*args)

Return or set pencolor and fillcolor.

Several input formats are allowed. They use 0 to 3 arguments as follows:

color() Return the current pencolor and the current fillcolor as a pair of color specification strings or tuples as returned by pencolor() and fillcolor().

color(colorstring), color((r,g,b)), color(r,g,b) Inputs as in pencolor(), set both, fillcolor and pencolor, to the given value.

color(colorstring1, colorstring2), color((r1,g1,b1), (r2,g2,b2))

Equivalent to pencolor(colorstring1) and fillcolor(colorstring2) and analogously if the other input format is used.

If turtleshape is a polygon, outline and interior of that polygon is drawn with the newly set colors.

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40, 80, 120), (160, 200, 240))
```

See also: Screen method colormode().

Filling

turtle.fill(flag)

Parameters flag – True/False (or 1/0 respectively)

Call fill(True) before drawing the shape you want to fill, and fill(False) when done. When used without argument: return fillstate (True if filling, False else).

```
>>> turtle.fill(True)
>>> for _ in range(3):
...     turtle.forward(100)
...     turtle.left(120)
...
>>> turtle.fill(False)
```

turtle.begin_fill()

Call just before drawing a shape to be filled. Equivalent to fill(True).

turtle.end_fill()
Fill the shape drawn after the last call to begin_fill(). Equivalent to fill(False).

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

More drawing control

turtle.reset()
Delete the turtle's drawings from the screen, re-center the turtle and set variables to the default values.

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.clear()
Delete the turtle's drawings from the screen. Do not move turtle. State and position of the turtle as well as drawings of other turtles are not affected.

turtle.write(arg, move=False, align="left", font=("Arial", 8, "normal"))

Parameters

- arg – object to be written to the TurtleScreen
- move – True/False
- align – one of the strings "left", "center" or right"
- font – a triple (fontname, fontsize, fonttype)

Write text - the string representation of arg - at the current turtle position according to align ("left", "center" or right") and with the given font. If move is true, the pen is moved to the bottom-right corner of the text. By default, move is False.

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

Turtle state

Visibility

turtle.hideturtle()
turtle.ht()
Make the turtle invisible. It's a good idea to do this while you're in the middle of doing some complex drawing, because hiding the turtle speeds up the drawing observably.

```
>>> turtle.hideturtle()
```

turtle.showturtle()
turtle.st()

Make the turtle visible.

```
>>> turtle.showturtle()
```

turtle.isvisible()

Return True if the Turtle is shown, False if it's hidden.

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

Appearance

turtle.shape(name=None)

Parameters name – a string which is a valid shapename

Set turtle shape to shape with given name or, if name is not given, return name of current shape. Shape with name must exist in the TurtleScreen's shape dictionary. Initially there are the following polygon shapes: "arrow", "turtle", "circle", "square", "triangle", "classic". To learn about how to deal with shapes see Screen method register_shape().

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.resizemode(rmode=None)

Parameters rmode – one of the strings "auto", "user", "noresize"

Set resizemode to one of the values: "auto", "user", "noresize". If rmode is not given, return current resizemode. Different resizemodes have the following effects:

- "auto": adapts the appearance of the turtle corresponding to the value of pensize.
- "user": adapts the appearance of the turtle according to the values of stretchfactor and outlinewidth (outline), which are set by shapesize().
- "noresize": no adaption of the turtle's appearance takes place.

resizemode("user") is called by shapesize() when used with arguments.

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.shapesize(stretch_wid=None, stretch_len=None, outline=None)

turtle.turtlesize(stretch_wid=None, stretch_len=None, outline=None)

Parameters

- stretch_wid – positive number
- stretch_len – positive number
- outline – positive number

Return or set the pen's attributes x/y-stretchfactors and/or outline. Set resizemode to "user". If and only if resizemode is set to "user", the turtle will be displayed stretched according to its stretchfactors: stretch_wid is stretchfactor perpendicular to its orientation, stretch_len is stretchfactor in direction of its orientation, outline determines the width of the shapes's outline.

```
>>> turtle.shapesize()
(1, 1, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.tilt(angle)

Parameters angle – a number

Rotate the turtleshape by angle from its current tilt-angle, but do not change the turtle's heading (direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.settiltangle(angle)

Parameters angle – a number

Rotate the turtleshape to point in the direction specified by angle, regardless of its current tilt-angle. Do not change the turtle's heading (direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

turtle.tiltangle()

Return the current tilt-angle, i.e. the angle between the orientation of the turtleshape and the heading of the turtle (its direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

Using events

turtle.onclick(fun, btn=1, add=None)

Parameters

- fun – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- btn – number of the mouse-button, defaults to 1 (left mouse button)
- add – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind fun to mouse-click events on this turtle. If fun is None, existing bindings are removed. Example for the anonymous turtle, i.e. the procedural way:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

turtle.onrelease(fun, btn=1, add=None)

Parameters

- fun – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- btn – number of the mouse-button, defaults to 1 (left mouse button)
- add – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind fun to mouse-button-release events on this turtle. If fun is None, existing bindings are removed.

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)      # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow) # releasing turns it to transparent.
```

turtle.ondrag(fun, btn=1, add=None)

Parameters

- fun – a function with two arguments which will be called with the coordinates of the clicked point on the canvas

- btn – number of the mouse-button, defaults to 1 (left mouse button)
- add – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind fun to mouse-move events on this turtle. If fun is None, existing bindings are removed.

Remark: Every sequence of mouse-move-events on a turtle is preceded by a mouse-click event on that turtle.

```
>>> turtle.ondrag(turtle.goto)
```

Subsequently, clicking and dragging the Turtle will move it across the screen thereby producing handdrawings (if pen is down).

turtle.mainloop()
turtle.done()
Starts event loop - calling Tkinter's mainloop function. Must be the last statement in a turtle graphics program.

```
>>> turtle.mainloop()
```

Special Turtle methods

turtle.begin_poly()
Start recording the vertices of a polygon. Current turtle position is first vertex of polygon.

turtle.end_poly()
Stop recording the vertices of a polygon. Current turtle position is last vertex of polygon. This will be connected with the first vertex.

turtle.get_poly()
Return the last recorded polygon.

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

turtle.clone()
Create and return a clone of the turtle with same position, heading and turtle properties.

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.getturtle()
turtle.getpen()
Return the Turtle object itself. Only reasonable use: as a function to return the "anonymous turtle":

```
>>> pet = getturtle()
>>> pet.fd(50)
```

(continues on next page)

(continued from previous page)

```
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.getscreen()
Return the TurtleScreen object the turtle is drawing on. TurtleScreen methods can then be called for that object.

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.setundobuffer(size)

Parameters size – an integer or None

Set or disable undobuffer. If size is an integer an empty undobuffer of given size is installed. size gives the maximum number of turtle actions that can be undone by the undo() method/function. If size is None, the undobuffer is disabled.

```
>>> turtle.setundobuffer(42)
```

turtle.undobufferentries()
Return number of entries in the undobuffer.

```
>>> while undobufferentries():
...     undo()
```

turtle.tracer(flag=None, delay=None)
A replica of the corresponding TurtleScreen method.

Deprecated since version 2.6.

turtle.window_width()
turtle.window_height()
Both are replicas of the corresponding TurtleScreen methods.

Deprecated since version 2.6.

Excursus about the use of compound shapes

To use compound turtle shapes, which consist of several polygons of different color, you must use the helper class Shape explicitly as described below:

1. Create an empty Shape object of type "compound".
2. Add as many components to this object as desired, using the addcomponent() method.

   For example:

```
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. Now add the Shape to the Screen's shapelist and use it:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

Note: The Shape class is used internally by the register_shape() method in different ways. The application programmer has to deal with the Shape class only when using compound shapes like shown above!

24.5.4 Methods of TurtleScreen/Screen and corresponding functions

Most of the examples in this section refer to a TurtleScreen instance called screen.

Window control

turtle.bgcolor(*args)

Parameters: args – a color string or three numbers in the range 0..colormode or a 3-tuple of such numbers

Set or return background color of the TurtleScreen.

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128, 0, 128)
```

turtle.bgpic(picname=None)

Parameters: picname – a string, name of a gif-file or "nopic", or None

Set background image or return name of current backgroundimage. If picname is a filename, set the corresponding image as background. If picname is "nopic", delete background image, if present. If picname is None, return the filename of the current backgroundimage.

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

turtle.clear()
turtle.clearscreen()

Delete all drawings and all turtles from the TurtleScreen. Reset the now empty TurtleScreen to its initial state: white background, no background image, no event bindings and tracing on.

Note: This TurtleScreen method is available as a global function only under the name clearscreen. The global function clear is another one derived from the Turtle method clear.

turtle.reset()
turtle.resetscreen()

Reset all Turtles on the Screen to their initial state.

Note: This TurtleScreen method is available as a global function only under the name resetscreen. The global function reset is another one derived from the Turtle method reset.

turtle.screensize(canvwidth=None, canvheight=None, bg=None)

Parameters

- canvwidth – positive integer, new width of canvas in pixels
- canvheight – positive integer, new height of canvas in pixels
- bg – colorstring or color-tuple, new background color

If no arguments are given, return current (canvaswidth, canvasheight). Else resize the canvas the turtles are drawing on. Do not alter the drawing window. To observe hidden parts of the canvas, use the scrollbars. With this method, one can make visible those parts of a drawing which were outside the canvas before.

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

e.g. to search for an erroneously escaped turtle ;-)

turtle.setworldcoordinates(llx, lly, urx, ury)

Parameters

- llx – a number, x-coordinate of lower left corner of canvas
- lly – a number, y-coordinate of lower left corner of canvas
- urx – a number, x-coordinate of upper right corner of canvas
- ury – a number, y-coordinate of upper right corner of canvas

Set up user-defined coordinate system and switch to mode "world" if necessary. This performs a screen.reset(). If mode "world" is already active, all drawings are redrawn according to the new coordinates.

ATTENTION: in user-defined coordinate systems angles may appear distorted.

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # a regular octagon
```

Animation control

turtle.delay(delay=None)

Parameters delay – positive integer

Set or return the drawing delay in milliseconds. (This is approximately the time interval between two consecutive canvas updates.) The longer the drawing delay, the slower the animation.

Optional argument:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.tracer(n=None, delay=None)

Parameters

- n – nonnegative integer
- delay – nonnegative integer

Turn turtle animation on/off and set delay for update drawings. If n is given, only each n-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.) Second argument sets delay value (see delay()).

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.update()

Perform a TurtleScreen update. To be used when tracer is turned off.

See also the RawTurtle/Turtle method speed().

Using screen events

turtle.listen(xdummy=None, ydummy=None)

Set focus on TurtleScreen (in order to collect key-events). Dummy arguments are provided in order to be able to pass listen() to the onclick method.

turtle.onkey(fun, key)

Parameters

- fun – a function with no arguments or None
- key – a string: key (e.g. “a”) or key-symbol (e.g. “space”)

Bind fun to key-release event of key. If fun is None, event bindings are removed. Remark: in order to be able to register key-events, TurtleScreen must have the focus. (See method listen().)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.onclick(fun, btn=1, add=None)
turtle.onscreenclick(fun, btn=1, add=None)

Parameters

- fun – a function with two arguments which will be called with the coordinates of the clicked point on the canvas

- btn – number of the mouse-button, defaults to 1 (left mouse button)
- add – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind fun to mouse-click events on this screen. If fun is None, existing bindings are removed.

Example for a TurtleScreen instance named screen and a Turtle instance named turtle:

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen will
>>>                              # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again
```

Note: This TurtleScreen method is available as a global function only under the name onscreenclick. The global function onclick is another one derived from the Turtle method onclick.

---

turtle.ontimer(fun, t=0)

Parameters

- fun – a function with no arguments
- t – a number >= 0

Install a timer that calls fun after t milliseconds.

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### makes the turtle march around
>>> running = False
```

Settings and special methods

turtle.mode(mode=None)

Parameters mode – one of the strings “standard”, “logo” or “world”

Set turtle mode (“standard”, “logo” or “world”) and perform reset. If mode is not given, current mode is returned.

Mode “standard” is compatible with old turtle. Mode “logo” is compatible with most Logo turtle graphics. Mode “world” uses user-defined “world coordinates”. Attention: in this mode angles appear distorted if x/y unit-ratio doesn’t equal 1.

Mode	Initial turtle heading	positive angles
“standard”	to the right (east)	counterclockwise
“logo”	upward (north)	clockwise

```
>>> mode("logo")   # resets turtle heading to north
>>> mode()
'logo'
```

turtle.colormode(cmode=None)

Parameters cmode – one of the values 1.0 or 255

Return the colormode or set it to 1.0 or 255. Subsequently r, g, b values of color triples have to be in the range 0..cmode.

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
    ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.getcanvas()

Return the Canvas of this TurtleScreen. Useful for insiders who know what to do with a Tkinter Canvas.

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas instance at 0x...>
```

turtle.getshapes()

Return a list of names of all currently available turtle shapes.

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.register_shape(name, shape=None)
turtle.addshape(name, shape=None)

There are three different ways to call this function:

(1) name is the name of a gif-file and shape is None: Install the corresponding image shape.

```
>>> screen.register_shape("turtle.gif")
```

Note: Image shapes do not rotate when turning the turtle, so they do not display the heading of the turtle!

(2) name is an arbitrary string and shape is a tuple of pairs of coordinates: Install the corresponding polygon shape.

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(3) name is an arbitrary string and shape is a (compound) Shape object: Install the corresponding compound shape.

Add a turtle shape to TurtleScreen's shapelist. Only thusly registered shapes can be used by issuing the command shape(shapename).

turtle.turtles()

Return the list of turtles on the screen.

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

turtle.window_height()
Return the height of the turtle window.

```
>>> screen.window_height()
480
```

turtle.window_width()
Return the width of the turtle window.

```
>>> screen.window_width()
640
```

Methods specific to Screen, not inherited from TurtleScreen

turtle.bye()
Shut the turtlegraphics window.

turtle.exitonclick()
Bind bye() method to mouse clicks on the Screen.

If the value "using_IDLE" in the configuration dictionary is False (default value), also enter mainloop. Remark: If IDLE with the -n switch (no subprocess) is used, this value should be set to True in turtle.cfg. In this case IDLE's own mainloop is active also for the client script.

turtle.setup(width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"])
Set the size and position of the main window. Default values of arguments are stored in the configuration dictionary and can be changed via a turtle.cfg file.

Parameters

- width – if an integer, a size in pixels, if a float, a fraction of the screen; default is 50% of screen
- height – if an integer, the height in pixels, if a float, a fraction of the screen; default is 75% of screen
- startx – if positive, starting position in pixels from the left edge of the screen, if negative from the right edge, if None, center window horizontally
- starty – if positive, starting position in pixels from the top edge of the screen, if negative from the bottom edge, if None, center window vertically

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>             # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>             # sets window to 75% of screen by 50% of screen and centers
```

turtle.title(titlestring)

Parameters titlestring – a string that is shown in the titlebar of the turtle graphics window

Set title of turtle window to titlestring.

```
>>> screen.title("Welcome to the turtle zoo!")
```

24.5.5 The public classes of the module turtle

class turtle.RawTurtle(canvas)
class turtle.RawPen(canvas)

Parameters canvas – a Tkinter.Canvas, a ScrolledCanvas or a TurtleScreen

Create a turtle. The turtle has all methods described above as "methods of Turtle/RawTurtle".

class turtle.Turtle
Subclass of RawTurtle, has the same interface but draws on a default Screen object created automatically when needed for the first time.

class turtle.TurtleScreen(cv)

Parameters cv – a Tkinter.Canvas

Provides screen oriented methods like setbg() etc. that are described above.

class turtle.Screen
Subclass of TurtleScreen, with four methods added.

class turtle.ScrolledCanvas(master)

Parameters master – some Tkinter widget to contain the ScrolledCanvas, i.e. a Tkinter-canvas with scrollbars added

Used by class Screen, which thus automatically provides a ScrolledCanvas as playground for the turtles.

class turtle.Shape(type_, data)

Parameters type_ – one of the strings "polygon", "image", "compound"

Data structure modeling shapes. The pair (type_, data) must follow this specification:

type_	data
"polygon"	a polygon-tuple, i.e. a tuple of pairs of coordinates
"image"	an image (in this form only used internally!)
"compound"	None (a compound shape has to be constructed using the addcomponent() method)

addcomponent(poly, fill, outline=None)

Parameters

- poly – a polygon, i.e. a tuple of pairs of numbers
- fill – a color the poly will be filled with
- outline – a color for the poly's outline (if given)

Example:

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

See Excursus about the use of compound shapes.

class turtle.Vec2D(x, y)
A two-dimensional vector class, used as a helper class for implementing turtle graphics. May be useful for turtle graphics programs too. Derived from tuple, so a vector is a tuple!

Provides (for a, b vectors, k number):

- a + b vector addition
- a - b vector subtraction
- a * b inner product
- k * a and a * k multiplication with scalar
- abs(a) absolute value of a
- a.rotate(angle) rotation

24.5.6 Help and configuration

How to use help

The public methods of the Screen and Turtle classes are documented extensively via docstrings. So these can be used as online-help via the Python help facilities:

- When using IDLE, tooltips show the signatures and first lines of the docstrings of typed in function-/method calls.
- Calling help() on methods or functions displays the docstrings:

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.


      >>> screen.bgcolor("orange")
      >>> screen.bgcolor()
      "orange"
      >>> screen.bgcolor(0.5,0,0.5)
      >>> screen.bgcolor()
      "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

- The docstrings of the functions which are derived from methods have a modified form:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
```

(continues on next page)

(continued from previous page)

```
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

These modified docstrings are created automatically together with the function definitions that are derived from the methods at import time.

Translation of docstrings into different languages

There is a utility to create a dictionary the keys of which are the method names and the values of which are the docstrings of the public methods of the classes Screen and Turtle.

turtle.write_docstringdict(filename="turtle_docstringdict")

Parameters filename – a string, used as filename

Create and write docstring-dictionary to a Python script with the given filename. This function has to be called explicitly (it is not used by the turtle graphics classes). The docstring dictionary will be written to the Python script filename.py. It is intended to serve as a template for translation of the docstrings into different languages.

If you (or your students) want to use turtle with online help in your native language, you have to translate the docstrings and save the resulting file as e.g. turtle_docstringdict_german.py.

If you have an appropriate entry in your turtle.cfg file this dictionary will be read in at import time and will replace the original English docstrings.

At the time of this writing there are docstring dictionaries in German and in Italian. (Requests please to glingl@aon.at.)

How to configure Screen and Turtles

The built-in default configuration mimics the appearance and behaviour of the old turtle module in order to retain best possible compatibility with it.

If you want to use a different configuration which better reflects the features of this module or which better fits to your needs, e.g. for use in a classroom, you can prepare a configuration file turtle.cfg which will be read at import time and modify the configuration according to its settings.

The built in configuration would correspond to the following turtle.cfg:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

Short explanation of selected entries:

- The first four lines correspond to the arguments of the Screen.setup() method.
- Line 5 and 6 correspond to the arguments of the method Screen.screensize().
- shape can be any of the built-in shapes, e.g: arrow, turtle, etc. For more info try help(shape).
- If you want to use no fillcolor (i.e. make the turtle transparent), you have to write fillcolor = "" (but all nonempty strings must not have quotes in the cfg-file).
- If you want to reflect the turtle its state, you have to use resizemode = auto.
- If you set e.g. language = italian the docstringdict turtle_docstringdict_italian.py will be loaded at import time (if present on the import path, e.g. in the same directory as turtle.
- The entries exampleturtle and examplescreen define the names of these objects as they occur in the docstrings. The transformation of method-docstrings to function-docstrings will delete these names from the docstrings.
- using_IDLE: Set this to True if you regularly work with IDLE and its -n switch ("no subprocess"). This will prevent exitonclick() to enter the mainloop.

There can be a turtle.cfg file in the directory where turtle is stored and an additional one in the current working directory. The latter will override the settings of the first one.

The Demo/turtle directory contains a turtle.cfg file. You can study it as an example and see its effects when running the demos (preferably not from within the demo-viewer).

24.5.7 Demo scripts

There is a set of demo scripts in the turtledemo directory located in the Demo/turtle directory in the source distribution.

It contains:

- a set of 15 demo scripts demonstrating different features of the new module turtle
- a demo viewer turtleDemo.py which can be used to view the sourcecode of the scripts and run them at the same time. 14 of the examples can be accessed via the Examples menu; all of them can also be run standalone.
- The example turtledemo_two_canvases.py demonstrates the simultaneous use of two canvases with the turtle module. Therefore it only can be run standalone.
- There is a turtle.cfg file in this directory, which also serves as an example for how to write and use such files.

The demoscripts are:

Name	Description	Features
bytedesign	complex classical turtlegraphics pattern	tracer(), delay, update()
chaos	graphs Verhulst dynamics, shows that computer's computations can generate results sometimes against the common sense expectations	world coordinates
clock	analog clock showing time of your computer	turtles as clock's hands, ontimer
colormixer	experiment with r, g, b	ondrag()
fractalcurves	Hilbert & Koch curves	recursion
lindenmayer	ethnomathematics (indian kolams)	L-System
minimal_hanoi	Towers of Hanoi	Rectangular Turtles as Hanoi discs (shape, shapesize)
paint	super minimalistic drawing program	onclick()
peace	elementary	turtle: appearance and animation
penrose	aperiodic tiling with kites and darts	stamp()
planet_and_moon	simulation of gravitational system	compound shapes, Vec2D
tree	a (graphical) breadth first tree (using generators)	clone()
wikipedia	a pattern from the wikipedia article on turtle graphics	clone(), undo()
yinyang	another elementary example	circle()

Have fun!

24.6 IDLE

IDLE is Python's Integrated Development and Learning Environment.

IDLE has the following features:

- coded in 100% pure Python, using the tkinter GUI toolkit
- cross-platform: works mostly the same on Windows, Unix, and Mac OS X

- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- search within any window, replace within editor windows, and search through multiple files (grep)
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces
- configuration, browsers, and other dialogs

24.6.1 Menus

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. Output windows, such as used for Edit / Find in Files, are a subtype of edit window. They currently have the same top menu as Editor windows but a different default title and context menu.

IDLE's menus dynamically change based on which window is currently selected. Each menu documented below indicates which window type it is associated with.

File menu (Shell and Editor)

New File Create a new file editing window.

Open... Open an existing file with an Open dialog.

Recent Files Open a list of recent files. Click one to open it.

Open Module... Open an existing module (searches sys.path).

Class Browser Show functions, classes, and methods in the current Editor file in a tree structure. In the shell, open a module first.

Path Browser Show sys.path directories, modules, functions, classes and methods in a tree structure.

Save Save the current window to the associated file, if there is one. Windows that have been changed since being opened or last saved have a * before and after the window title. If there is no associated file, do Save As instead.

Save As... Save the current window with a Save As dialog. The file saved becomes the new associated file for the window.

Save Copy As... Save the current window to different file without changing the associated file.

Print Window Print the current window to the default printer.

Close Close the current window (ask to save if unsaved).

Exit Close all windows and quit IDLE (ask to save unsaved windows).

Edit menu (Shell and Editor)

Undo Undo the last change to the current window. A maximum of 1000 changes may be undone.

Redo Redo the last undone change to the current window.

Cut Copy selection into the system-wide clipboard; then delete the selection.

Copy Copy selection into the system-wide clipboard.

Paste Insert contents of the system-wide clipboard into the current window.

The clipboard functions are also available in context menus.

Select All Select the entire contents of the current window.

Find... Open a search dialog with many options

Find Again Repeat the last search, if there is one.

Find Selection Search for the currently selected string, if there is one.

Find in Files... Open a file search dialog. Put results in a new output window.

Replace... Open a search-and-replace dialog.

Go to Line Move cursor to the line number requested and make that line visible.

Show Completions Open a scrollable list allowing selection of keywords and attributes. See Completions in the Tips sections below.

Expand Word Expand a prefix you have typed to match a full word in the same window; repeat to get a different expansion.

Show call tip After an unclosed parenthesis for a function, open a small window with function parameter hints.

Show surrounding parens Highlight the surrounding parenthesis.

Format menu (Editor window only)

Indent Region Shift selected lines right by the indent width (default 4 spaces).

Dedent Region Shift selected lines left by the indent width (default 4 spaces).

Comment Out Region Insert ## in front of selected lines.

Uncomment Region Remove leading # or ## from selected lines.

Tabify Region Turn leading stretches of spaces into tabs. (Note: We recommend using 4 space blocks to indent Python code.)

Untabify Region Turn all tabs into the correct number of spaces.

Toggle Tabs Open a dialog to switch between indenting with spaces and tabs.

New Indent Width Open a dialog to change indent width. The accepted default by the Python community is 4 spaces.

Format Paragraph Reformat the current blank-line-delimited paragraph in comment block or multiline string or selected line in a string. All lines in the paragraph will be formatted to less than N columns, where N defaults to 72.

Strip trailing whitespace Remove any space characters after the last non-space character of a line.

Run menu (Editor window only)

Python Shell Open or wake up the Python Shell window.

Check Module Check the syntax of the module currently open in the Editor window. If the module has not been saved IDLE will either prompt the user to save or autosave, as selected in the General tab of the Idle Settings dialog. If there is a syntax error, the approximate location is indicated in the Editor window.

Run Module Do Check Module (above). If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of print or write. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with python -i file at a command line.

Shell menu (Shell window only)

View Last Restart Scroll the shell window to the last Shell restart.

Restart Shell Restart the shell to clean the environment.

Interrupt Execution Stop a running program.

Debug menu (Shell window only)

Go to File/Line Look on the current line. with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.

Debugger (toggle) When activated, code entered in the Shell or run from an Editor will run under the debugger. In the Editor, breakpoints can be set with the context menu. This feature is still incomplete and somewhat experimental.

Stack Viewer Show the stack traceback of the last exception in a tree widget, with access to locals and globals.

Auto-open Stack Viewer Toggle automatically opening the stack viewer on an unhandled exception.

Options menu (Shell and Editor)

Configure IDLE Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions (see below). On OS X, open the configuration dialog by selecting Preferences in the application menu. To use a new built-in color theme (IDLE Dark) with older IDLEs, save it as a new custom theme.

Non-default user settings are saved in a .idlerc directory in the user's home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in .idlerc.

Code Context (toggle)(Editor Window only) Open a pane at the top of the edit window which shows the block context of the code which has scrolled above the top of the window.

Window menu (Shell and Editor)

Zoom Height Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog.

The rest of this menu lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

Help menu (Shell and Editor)

About IDLE Display version, copyright, license, credits, and more.

IDLE Help Display a help file for IDLE detailing the menu options, basic editing and navigation, and other tips.

Python Docs Access local Python documentation, if installed, or start a web browser and open docs.python.org showing the latest Python documentation.

Turtle Demo Run the turtledemo module with example python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab.

Context Menus

Open a context menu by right-clicking in a window (Control-click on OS X). Context menus have the standard clipboard functions also on the Edit menu.

Cut Copy selection into the system-wide clipboard; then delete the selection.

Copy Copy selection into the system-wide clipboard.

Paste Insert contents of the system-wide clipboard into the current window.

Editor windows also have breakpoint functions. Lines with a breakpoint set are specially marked. Breakpoints only have an effect when running under the debugger. Breakpoints for a file are saved in the user's .idlerc directory.

Set Breakpoint Set a breakpoint on the current line.

Clear Breakpoint Clear the breakpoint on that line.

Shell and Output windows have the following.

Go to file/line Same as in Debug menu.

24.6.2 Editing and navigation

In this section, 'C' refers to the Control key on Windows and Unix and the Command key on Mac OSX.

- Backspace deletes to the left; Del deletes to the right
- C-Backspace delete word left; C-Del delete word to the right
- Arrow keys and Page Up/Page Down to move around
- C-LeftArrow and C-RightArrow moves by words
- Home/End go to begin/end of line
- C-Home/C-End go to begin/end of file
- Some useful Emacs bindings are inherited from Tcl/Tk:
  - C-a beginning of line
  - C-e end of line
  - C-k kill line (but doesn't put it in clipboard)
  - C-l center window around the insertion point
  - C-b go backward one character without deleting (usually you can also use the cursor key for this)

- C-f go forward one character without deleting (usually you can also use the cursor key for this)
- C-p go up one line (usually you can also use the cursor key for this)
- C-d delete next character

Standard keybindings (like C-c to copy and C-v to paste) may work. Keybindings are selected in the Configure IDLE dialog.

Automatic indentation

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, Backspace deletes up to 4 spaces if they are there. Tab inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently, tabs are restricted to four spaces due to Tcl/Tk limitations.

See also the indent/dedent region commands in the edit menu.

Completions

Completions are supplied for functions, classes, and attributes of classes, both built-in and user-defined. Completions are also provided for filenames.

The AutoCompleteWindow (ACW) will open after a predefined delay (default is two seconds) after a '.' or (in a string) an os.sep is typed. If after one of those characters (plus zero or more other characters) a tab is typed the ACW will open immediately if a possible continuation is found.

If there is only one possible completion for the characters entered, a Tab will supply that completion without opening the ACW.

'Show Completions' will force open a completions window, by default the C-space will open a completions window. In an empty string, this will contain the files in the current directory. On a blank line, it will contain the built-in and user-defined functions and classes in the current namespaces, plus any modules imported. If some characters have been entered, the ACW will attempt to be more specific.

If a string of characters is typed, the ACW selection will jump to the entry most closely matching those characters. Entering a tab will cause the longest non-ambiguous match to be entered in the Editor window or Shell. Two tab in a row will supply the current ACW selection, as will return or a double click. Cursor keys, Page Up/Down, mouse selection, and the scroll wheel all operate on the ACW.

"Hidden" attributes can be accessed by typing the beginning of hidden name after a '.', e.g. '_'. This allows access to modules with __all__ set, or to class-private attributes.

Completions and the 'Expand Word' facility can save a lot of typing!

Completions are currently limited to those in the namespaces. Names in an Editor window which are not via __main__ and sys.modules will not be found. Run the module once with your imports to correct this situation. Note that IDLE itself places quite a few modules in sys.modules, so much can be found by default, e.g. the re module.

If you don't like the ACW popping up unbidden, simply make the delay longer or disable the extension.

Calltips

A calltip is shown when one types (after the name of an accessible function. A name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or) is typed. When the cursor is in the argument part of a definition, the menu or shortcut display a calltip.

A calltip consists of the function signature and the first line of the docstring. For builtins without an accessible signature, the calltip consists of all lines up the fifth line or the first blank line. These details may change.

The set of accessible functions depends on what modules have been imported into the user process, including those imported by Idle itself, and what definitions have been run, all since the last restart.

For example, restart the Shell and enter itertools.count(. A calltip appears because Idle imports itertools into the user process for its own use. (This could change.) Enter turtle.write(and nothing appears. Idle does not import turtle. The menu or shortcut do nothing either. Enter import turtle and then turtle.write(will work.

In an editor, import statements have no effect until one runs the file. One might want to run a file after writing the import statements at the top, or immediately run an existing file before editing.

Python Shell window

- C-c interrupts executing command
- C-d sends end-of-file; closes window if typed at a >>> prompt
- Alt-/ (Expand word) is also useful to reduce typing

  Command history

  - Alt-p retrieves previous command matching what you have typed. On OS X use C-p.
  - Alt-n retrieves next. On OS X use C-n.
  - Return while on any previous command retrieves that command

Text colors

Idle defaults to black on white text, but colors text with special meanings. For the shell, these are shell output, shell error, user output, and user error. For Python code, at the shell prompt or in an editor, these are keywords, builtin class and function names, names following class and def, strings, and comments. For any text window, these are the cursor (when present), found text (when possible), and selected text.

Text coloring is done in the background, so uncolorized text is occasionally visible. To change the color scheme, use the Configure IDLE dialog Highlighting tab. The marking of debugger breakpoint lines in the editor and text in popups and dialogs is not user-configurable.

24.6.3 Startup and code execution

Upon startup with the -s option, IDLE will execute the file referenced by the environment variables IDLESTARTUP or PYTHONSTARTUP. IDLE first checks for IDLESTARTUP; if IDLESTARTUP is present the file referenced is run. If IDLESTARTUP is not present, IDLE checks for PYTHONSTARTUP. Files referenced by these environment variables are convenient places to store functions that are used frequently from the IDLE shell, or for executing import statements to import common modules.

In addition, Tk also loads a startup file if it is present. Note that the Tk file is loaded unconditionally. This additional file is .Idle.py and is looked for in the user's home directory. Statements in this file will be executed in the Tk namespace, so this file is not useful for importing functions to be used from IDLE's Python shell.

Command line usage

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command  run command in the shell window
-d          enable debugger and open shell window
-e          open editor window
-h          print help message with legal combinations and exit
-i          open shell window
-r file     run file in shell window
-s          run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title    set title of shell window
-           run stdin in shell (- must be last option before args)
```

If there are arguments:

- If -, -c, or r is used, all arguments are placed in sys.argv[1:...] and sys.argv[0] is set to `''`, `'-c'`, or `'-r'`. No editor window is opened, even if that is the default set in the Options dialog.
- Otherwise, arguments are files opened for editing and sys.argv reflects the arguments passed to IDLE itself.

IDLE-console differences

As much as possible, the result of executing Python code with IDLE is the same as executing the same code in a console window. However, the different interface and operation occasionally affect visible results. For instance, sys.modules starts with more entries.

IDLE also replaces sys.stdin, sys.stdout, and sys.stderr with objects that get input from and send output to the Shell window. When this window has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. If sys is reset with reload(sys), IDLE's changes are lost and things like input, raw_input, and print will not work correctly.

With IDLE's Shell, one enters, edits, and recalls complete statements. Some consoles only work with a single physical line at a time. IDLE uses exec to run each statement. As a result, `'__builtins__'` is always defined for each statement.

Running without a subprocess

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the Internet. If firewall software complains anyway, you can ignore it.

If the attempt to make the socket connection fails, Idle will notify you. Such failures are sometimes transient, but if persistent, the problem may be either a firewall blocking the connection or misconfiguration of a particular system. Until the problem is fixed, one can run Idle with the -n command line switch.

If IDLE is started with the -n command line switch it will run in a single process and will not create the subprocess which runs the RPC Python execution server. This can be useful if Python cannot create the subprocess or the RPC socket interface on your platform. However, in this mode user code is not isolated from IDLE itself. Also, the environment is not restarted when Run/Run Module (F5) is selected. If your code has been modified, you must reload() the affected modules and re-import any specific items (e.g. from foo import baz) if the changes are to take effect. For these reasons, it is preferable to run IDLE with the default subprocess if at all possible.

Deprecated since version 3.4.

24.6.4 Help and preferences

Additional help sources

IDLE includes a help menu entry called "Python Docs" that will open the extensive sources of help, including tutorials, available at docs.python.org. Selected URLs can be added or removed from the help menu at any time using the Configure IDLE dialog. See the IDLE help option in the help menu of IDLE for more information.

Setting preferences

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Keys can be user defined; IDLE ships with four built-in key sets. In addition, a user can create a custom key set in the Configure IDLE dialog under the keys tab.

Extensions

IDLE contains an extension facility. Preferences for extensions can be changed with Configure Extensions. See the beginning of config-extensions.def in the idlelib directory for further information. The default extensions are currently:

- FormatParagraph
- AutoExpand
- ZoomHeight
- ScriptBinding
- CallTips
- ParenMatch
- AutoComplete
- CodeContext
- RstripExtension

24.7 Other Graphical User Interface Packages

Major cross-platform (Windows, Mac OS X, Unix-like) GUI toolkits are available for Python:

See also:

PyGTK is a set of bindings for the GTK widget set. It provides an object oriented interface that is slightly higher level than the C one. It comes with many more widgets than Tkinter provides, and has good Python-specific reference documentation. There are also bindings to GNOME. An online tutorial is available.

PyQt PyQt is a sip-wrapped binding to the Qt toolkit. Qt is an extensive C++ GUI application development framework that is available for Unix, Windows and Mac OS X. sip is a tool for generating bindings for C++ libraries as Python classes, and is specifically designed for Python. The PyQt3 bindings have a book, GUI Programming with Python: QT Edition by Boudewijn Rempt. The PyQt4 bindings also have a book, Rapid GUI Programming with Python and Qt, by Mark Summerfield.

wxPython wxPython is a cross-platform GUI toolkit for Python that is built around the popular wxWidgets (formerly wxWindows) C++ toolkit. It provides a native look and feel for applications on Windows, Mac OS X, and Unix systems by using each platform's native widgets where ever possible, (GTK+ on Unix-like systems). In addition to an extensive set of widgets, wxPython provides classes for online documentation and context sensitive help, printing, HTML viewing, low-level device context drawing, drag and drop, system clipboard access, an XML-based resource format and more, including an ever growing library of user-contributed modules. wxPython has a book, wxPython in Action, by Noel Rappin and Robin Dunn.

PyGTK, PyQt, and wxPython, all have a modern look and feel and more widgets than Tkinter. In addition, there are many other GUI toolkits for Python, both cross-platform, and platform-specific. See the GUI Programming page in the Python Wiki for a much more complete list, and also for links to documents where the different GUI toolkits are compared.

CHAPTER

TWENTYFIVE

DEVELOPMENT TOOLS

The modules described in this chapter help you write software. For example, the pydoc module takes a module and generates documentation based on the module's contents. The doctest and unittest modules contains frameworks for writing unit tests that automatically exercise code and verify that the expected output is produced. 2to3 can translate Python 2.x source code into valid Python 3.x code.

The list of modules described in this chapter is:

25.1 pydoc — Documentation generator and online help system

New in version 2.1.

Source code: Lib/pydoc.py

The pydoc module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.

For modules, classes, functions and methods, the displayed documentation is derived from the docstring (i.e. the __doc__ attribute) of the object, and recursively of its documentable members. If there is no docstring, pydoc tries to obtain a description from the block of comment lines just above the definition of the class, function or method in the source file, or at the top of the module (see inspect.getcomments()).

The built-in function help() invokes the online help system in the interactive interpreter, which uses pydoc to generate its documentation as text on the console. The same text documentation can also be viewed from outside the Python interpreter by running pydoc as a script at the operating system's command prompt. For example, running

```
pydoc sys
```

at a shell prompt will display documentation on the sys module, in a style similar to the manual pages shown by the Unix man command. The argument to pydoc can be the name of a function, module, or package, or a dotted reference to a class, method, or function within a module or module in a package. If the argument to pydoc looks like a path (that is, it contains the path separator for your operating system, such as a slash in Unix), and refers to an existing Python source file, then documentation is produced for that file.

Note: In order to find objects and their documentation, pydoc imports the module(s) to be documented. Therefore, any code on module level will be executed on that occasion. Use an if __name__ == '__main__': guard to only execute code when a file is invoked as a script and not just imported.

When printing output to the console, pydoc attempts to paginate the output for easier reading. If the PAGER environment variable is set, pydoc will use its value as a pagination program.

Specifying a -w flag before the argument will cause HTML documentation to be written out to a file in the current directory, instead of displaying text on the console.

Specifying a -k flag before the argument will search the synopsis lines of all available modules for the keyword given as the argument, again in a manner similar to the Unix man command. The synopsis line of a module is the first line of its documentation string.

You can also use pydoc to start an HTTP server on the local machine that will serve documentation to visiting Web browsers. pydoc -p 1234 will start a HTTP server on port 1234, allowing you to browse the documentation at http://localhost:1234/ in your preferred Web browser. pydoc -g will start the server and additionally bring up a small Tkinter-based graphical interface to help you search for documentation pages.

When pydoc generates documentation, it uses the current environment and path to locate modules. Thus, invoking pydoc spam documents precisely the version of the module you would get if you started the Python interpreter and typed import spam.

Module docs for core modules are assumed to reside in https://docs.python.org/library/. This can be overridden by setting the PYTHONDOCS environment variable to a different URL or to a local directory containing the Library Reference Manual pages.

25.2 doctest — Test interactive Python examples

The doctest module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest:

- To check that a module's docstrings are up-to-date by verifying that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of "literate testing" or "executable documentation".

Here's a complete but small example module:

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int, return an int.
    Else return a long.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
```

(continues on next page)

(continued from previous page)

```
    265252859812191058636308480000000L
    >>> factorial(30L)
    265252859812191058636308480000000L
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000L

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

If you run example.py directly from the command line, doctest works its magic:

```
$ python example.py
$
```

There's no output! That's normal, and it means all the examples worked. Pass -v to the script, and doctest prints a detailed log of what it's trying, and prints a summary at the end:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
```

(continues on next page)

(continued from previous page)

```
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
Trying:
    [factorial(long(n)) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

And so on, eventually ending with:

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 tests in __main__
   8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$
```

That's all you need to know to start making productive use of doctest! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially useful examples can be found in the standard test file Lib/test/test_doctest.py.

25.2.1 Simple Usage: Checking Examples in Docstrings

The simplest way to start using doctest (but not necessarily the way you'll continue to do it) is to end each module M with:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

doctest then examines docstrings in module M.

Running the module as a script causes the examples in the docstrings to get executed and verified:

```
python M.py
```

This won't display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is ***Test Failed*** N failures., where N is the number of examples that failed.

Run it with the -v switch instead:

```
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing verbose=True to testmod(), or prohibit it by passing verbose=False. In either of those cases, sys.argv is not examined by testmod() (so passing -v or not has no effect).

Since Python 2.6, there is also a command line shortcut for running testmod(). You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the module name(s) on the command line:

```
python -m doctest -v example.py
```

This will import example.py as a standalone module and run testmod() on it. Note that this may not work correctly if the file is part of a package and imports other submodules from that package.

For more information on testmod(), see section Basic API.

25.2.2 Simple Usage: Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the testfile() function:

```
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file example.txt. The file content is treated as if it were a single giant docstring; the file doesn't need to contain a Python program! For example, perhaps example.txt contains this:

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

Running doctest.testfile("example.txt") then finds the error in this documentation:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

As with testmod(), testfile() won't display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as testmod().

By default, testfile() looks for files in the calling module's directory. See section Basic API for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like testmod(), testfile()'s verbosity can be set with the -v command-line switch or with the optional keyword argument verbose.

Since Python 2.6, there is also a command line shortcut for running testfile(). You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the file name(s) on the command line:

```
python -m doctest -v example.txt
```

Because the file name does not end with .py, doctest infers that it must be run with testfile(), not testmod().

For more information on testfile(), see section Basic API.

25.2.3 How It Works

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

Which Docstrings Are Examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

In addition, if M.__test__ exists and "is true", it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from M.__test__ are searched, and strings are treated as if they were docstrings. In output, a key K in M.__test__ appears with name

```
<name of M>.__test__.K
```

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.

Changed in version 2.4: A "private name" concept is deprecated and no longer documented.

How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn't trying to do an exact emulation of any specific Python shell.

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print "yes"
... else:
...     print "no"
...     print "NO"
...     print "NO!!!"
```

(continues on next page)

(continued from previous page)

```
...
no
NO
NO!!!
>>>
```

Any expected output must immediately follow the final `'>>> '` or `'... '` line containing the code, and the expected output (if any) extends to the next `'>>> '` or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put <BLANKLINE> in your doctest example each place a blank line is expected.

  New in version 2.4: <BLANKLINE> was added; there was no way to use expected output containing empty lines in previous versions.

- All hard tab characters are expanded to spaces, using 8-column tab stops. Tabs in output generated by the tested code are not modified. Because any hard tabs in the sample output are expanded, this means that if the code output includes hard tabs, the only way the doctest can pass is if the NORMALIZE_WHITESPACE option or directive is in effect. Alternatively, the test can be rewritten to capture the output and compare it to an expected value as part of the test. This handling of tabs in the source was arrived at through trial and error, and has proven to be the least error prone way of handling them. It is possible to use a different algorithm for handling tabs by writing a custom DocTestParser class.

  Changed in version 2.4: Expanding tabs to spaces is new; previous versions tried to preserve hard tabs, with confusing results.

- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).

- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them:

  ```
  >>> def f(x):
  ...     r'''Backslashes in a raw docstring: m\n'''
  >>> print f.__doc__
  Backslashes in a raw docstring: m\n
  ```

  Otherwise, the backslash will be interpreted as part of the string. For example, the \n above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string):

  ```
  >>> def f(x):
  ...     '''Backslashes in a raw docstring: m\\n'''
  >>> print f.__doc__
  Backslashes in a raw docstring: m\n
  ```

- The starting column doesn't matter:

  ```
  >>> assert "Easy!"
      >>> import math
          >>> math.floor(1.9)
          1.0
  ```

and as many leading whitespace characters are stripped from the expected output as appeared in the initial '>>> ' line that started the example.

What's the Execution Context?

By default, each time doctest finds a docstring to test, it uses a shallow copy of M's globals, so that running tests doesn't change the module's real globals, and so that one test in M can't leave behind crumbs that accidentally allow another test to work. This means examples can freely use any names defined at top-level in M, and names defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own dict as the execution context by passing globs=your_dict to testmod() or testfile() instead.

What About Exceptions?

No problem, provided that the traceback is the only output produced by the example: just paste in the traceback.[1] Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers), this is one case where doctest works hard to be flexible in what it accepts.

Simple example:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

That doctest succeeds if ValueError is raised, with the list.remove(x): x not in list detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example:

```
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

The last three lines (starting with ValueError) are compared against the exception's type and detail, and the rest are ignored.

Changed in version 2.4: Previous versions were unable to handle multi-line exception details.

[1] Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is too error-prone, and that also makes for a confusing test.

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of ... is independent of doctest's ELLIPSIS option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won't need to remember:

- Doctest can't guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects ValueError: 42 is prime will pass whether ValueError is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn't create real problems.
- Each line of the traceback stack (if present) must be indented further than the first line of the example, or start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.
- When the IGNORE_EXCEPTION_DETAIL doctest option is specified, everything following the leftmost colon and any module information in the exception name is ignored.
- The interactive shell omits the traceback header line for some SyntaxErrors. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a SyntaxError that omits the traceback header, you will need to manually add the traceback header line to your test example.
- For some SyntaxErrors, Python displays the character position of the syntax error, using a ^ marker:

  ```
  >>> 1 1
    File "<stdin>", line 1
      1 1
        ^
  SyntaxError: invalid syntax
  ```

  Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the ^ marker in the wrong location:

  ```
  >>> 1 1
    File "<stdin>", line 1
      1 1
      ^
  SyntaxError: invalid syntax
  ```

Option Flags

A number of option flags control various aspects of doctest's behavior. Symbolic names for the flags are supplied as module constants, which can be bitwise ORed together and passed to various functions. The names can also be used in doctest directives.

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example's expected output:

doctest.DONT_ACCEPT_TRUE_FOR_1
: By default, if an expected output block contains just 1, an actual output block containing just 1 or just True is considered to be a match, and similarly for 0 versus False. When DONT_ACCEPT_TRUE_FOR_1 is specified, neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting "little integer" output still work in these cases. This option will probably go away, but not for several years.

doctest.DONT_ACCEPT_BLANKLINE
: By default, if an expected output block contains a line containing only the string <BLANKLINE>, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output, this is the only way to communicate that a blank line is expected. When DONT_ACCEPT_BLANKLINE is specified, this substitution is not allowed.

doctest.NORMALIZE_WHITESPACE
: When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. NORMALIZE_WHITESPACE is especially useful when a line of expected output is very long, and you want to wrap it across multiple lines in your source.

doctest.ELLIPSIS
: When specified, an ellipsis marker (...) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it's best to keep usage of this simple. Complicated uses can lead to the same kinds of "oops, it matched too much!" surprises that .* is prone to in regular expressions.

doctest.IGNORE_EXCEPTION_DETAIL
: When specified, an example that expects an exception passes if an exception of the expected type is raised, even if the exception detail does not match. For example, an example expecting ValueError: 42 will pass if the actual exception raised is ValueError: 3*14, but will fail, e.g., if TypeError is raised.

  It will also ignore the module name used in Python 3 doctest reports. Hence both of these variations will work with the flag specified, regardless of whether the test is run under Python 2.7 or Python 3.2 (or later versions):

```
>>> raise CustomError('message')
Traceback (most recent call last):
CustomError: message

>>> raise CustomError('message')
Traceback (most recent call last):
my_module.CustomError: message
```

  Note that ELLIPSIS can also be used to ignore the details of the exception message, but such a test may still fail based on whether or not the module details are printed as part of the exception name. Using IGNORE_EXCEPTION_DETAIL and the details from Python 2.3 is also the only clear way to write a doctest that doesn't care about the exception detail yet continues to pass under Python 2.3 or earlier (those releases do not support doctest directives and ignore them as irrelevant comments). For example:

```
>>> (1, 2)[3] = 'moo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object doesn't support item assignment
```

passes under Python 2.3 and later Python versions with the flag specified, even though the detail changed in Python 2.4 to say "does not" instead of "doesn't".

Changed in version 2.7: IGNORE_EXCEPTION_DETAIL now also ignores any information relating to the module containing the exception under test

doctest.SKIP
When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both documentation and test cases, and an example should be included for documentation purposes, but should not be checked. E.g., the example's output might be random; or the example might depend on resources which would be unavailable to the test driver.

The SKIP flag can also be used for temporarily "commenting out" examples.

New in version 2.5.

doctest.COMPARISON_FLAGS
A bitmask or'ing together all the comparison flags above.

The second group of options controls how test failures are reported:

doctest.REPORT_UDIFF
When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

doctest.REPORT_CDIFF
When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

doctest.REPORT_NDIFF
When specified, differences are computed by difflib.Differ, using the same algorithm as the popular ndiff.py utility. This is the only method that marks differences within lines as well as across lines. For example, if a line of expected output contains digit 1 where actual output contains letter l, a line is inserted with a caret marking the mismatching column positions.

doctest.REPORT_ONLY_FIRST_FAILURE
When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When REPORT_ONLY_FIRST_FAILURE is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

doctest.REPORTING_FLAGS
A bitmask or'ing together all the reporting flags above.

New in version 2.4: The constants DONT_ACCEPT_BLANKLINE, NORMALIZE_WHITESPACE, ELLIPSIS, IGNORE_EXCEPTION_DETAIL, REPORT_UDIFF, REPORT_CDIFF, REPORT_NDIFF, REPORT_ONLY_FIRST_FAILURE, COMPARISON_FLAGS and REPORTING_FLAGS were added.

There's also a way to register new option flag names, although this isn't useful unless you intend to extend doctest internals via subclassing:

doctest.register_optionflag(name)
Create a new option flag with a given name, and return the new flag's integer value. register_optionflag() can be used when subclassing OutputChecker or DocTestRunner to create new options that are supported by your subclasses. register_optionflag() should always be called using the following idiom:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

New in version 2.4.

Directives

Doctest directives may be used to modify the option flags for an individual example. Doctest directives are special Python comments following an example's source code:

```
directive             ::=  "#" "doctest:" directive_options
directive_options     ::=  directive_option ("," directive_option)\*
directive_option      ::=  on_or_off directive_option_name
on_or_off             ::=  "+" \| "-"
directive_option_name ::=  "DONT_ACCEPT_BLANKLINE" \| "NORMALIZE_WHITESPACE" \| ...
```

Whitespace is not allowed between the + or - and the directive option name. The directive option name can be any of the option flag names explained above.

An example's doctest directives modify doctest's behavior for that single example. Use + to enable the named behavior, or - to disable it.

For example, this test passes:

```
>>> print range(20)
[0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Without the directive it would fail, both because the actual output doesn't have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so:

```
>>> print range(20)
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas:

```
>>> print range(20)
[0,    1, ...,   18,    19]
```

If multiple directive comments are used for a single example, then they are combined:

```
>>> print range(20)
...
[0,    1, ...,   18,    19]
```

As the previous example shows, you can add ... lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line:

```
>>> print range(5) + range(10,20) + range(30,40) + range(50,60)
...
[0, ..., 4, 10, ..., 19, 30, ..., 39, 50, ..., 59]
```

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via + in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via - in a directive can be useful.

New in version 2.4: Support for doctest directives was added.

Warnings

doctest is serious about requiring exact matches in expected output. If even a single character doesn't match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn't guarantee about output. For example, when printing a dict, Python doesn't guarantee that the key-value pairs will be printed in any particular order, so a test like

```
>>> foo()
{"Hermione": "hippogryph", "Harry": "broomstick"}
```

is vulnerable! One workaround is to do

```
>>> foo() == {"Hermione": "hippogryph", "Harry": "broomstick"}
True
```

instead. Another is to do

```
>>> d = foo().items()
>>> d.sort()
>>> d
[('Harry', 'broomstick'), ('Hermione', 'hippogryph')]
```

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

```
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C()  # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

The ELLIPSIS directive gives a nice approach for the last example:

```
>>> C()
<__main__.C instance at 0x...>
```

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for float formatting, and C libraries vary widely in quality here.

```
>>> 1./7  # risky
0.14285714285714285
>>> print 1./7 # safer
0.142857142857
>>> print round(1./7, 6) # much safer
0.142857
```

Numbers of the form I/2.**J are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

```
>>> 3./4  # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.

25.2.4 Basic API

The functions testmod() and testfile() provide a simple interface to doctest that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections Simple Usage: Checking Examples in Docstrings and Simple Usage: Checking Examples in a Text File.

doctest.testfile(filename[, module_relative][, name][, package][, globs][, verbose][, report][, optionflags][, extraglobs][, raise_on_error][, parser][, encoding])

All arguments except filename are optional, and should be specified in keyword form.

Test examples in the file named filename. Return (failure_count, test_count).

Optional argument module_relative specifies how the filename should be interpreted:

- If module_relative is True (the default), then filename specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the package argument is specified, then it is relative to that package. To ensure OS-independence, filename should use / characters to separate path segments, and may not be an absolute path (i.e., it may not begin with /).
- If module_relative is False, then filename specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument name gives the name of the test; by default, or if None, os.path.basename(filename) is used.

Optional argument package is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify package if module_relative is False.

Optional argument globs gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if None, a new empty dict is used.

Optional argument extraglobs gives a dict merged into the globals used to execute examples. This works like dict.update(): if globs and extraglobs have a common key, the associated value in extraglobs appears in the combined dict. By default, or if None, no extra globals are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an extraglobs dict mapping the generic name to the subclass to be tested.

Optional argument verbose prints lots of stuff if true, and prints only failures if false; by default, or if None, it's true if and only if `'-v'` is in sys.argv.

Optional argument report prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).

Optional argument optionflags or's together option flags. See section Option Flags.

Optional argument raise_on_error defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument parser specifies a DocTestParser (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., DocTestParser()).

Optional argument encoding specifies an encoding that should be used to convert the file to unicode.

New in version 2.4.

Changed in version 2.5: The parameter encoding was added.

doctest.testmod([m][, name][, globs][, verbose][, report][, optionflags][, extraglobs][, raise_on_error][, exclude_empty])

All arguments are optional, and all except for m should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module m (or module __main__ if m is not supplied or is None), starting with m.__doc__.

Also test examples reachable from dict m.__test__, if it exists and is not None. m.__test__ maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only docstrings attached to objects belonging to module m are searched.

Return (failure_count, test_count).

Optional argument name gives the name of the module; by default, or if None, m.__name__ is used.

Optional argument exclude_empty defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using doctest.master.summarize() in conjunction with testmod() continues to get output for objects with no tests. The exclude_empty argument to the newer DocTestFinder constructor defaults to true.

Optional arguments extraglobs, verbose, report, optionflags, raise_on_error, and globs are the same as for function testfile() above, except that globs defaults to m.__dict__.

Changed in version 2.3: The parameter optionflags was added.

Changed in version 2.4: The parameters extraglobs, raise_on_error and exclude_empty were added.

Changed in version 2.5: The optional argument isprivate, deprecated in 2.4, was removed.

doctest.run_docstring_examples(f, globs[, verbose][, name][, compileflags][, optionflags])

Test examples associated with object f; for example, f may be a string, a module, a function, or a class object.

A shallow copy of dictionary argument globs is used for the execution context.

Optional argument name is used in failure messages, and defaults to "NoName".

If optional argument verbose is true, output is generated even if there are no failures. By default, output is generated only in case of an example failure.

Optional argument compileflags gives the set of flags that should be used by the Python compiler when running the examples. By default, or if None, flags are deduced corresponding to the set of future features found in globs.

Optional argument optionflags works as for function testfile() above.

25.2.5 Unittest API

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. Prior to Python 2.4, doctest had a barely documented Tester class that supplied a rudimentary way to combine doctests from multiple modules. Tester was feeble, and in practice most serious Python testing frameworks build on the unittest module, which supplies many flexible ways to combine tests from multiple sources. So, in Python 2.4, doctest's Tester class is deprecated, and doctest provides two functions that can be used to create unittest test suites from modules and text files containing doctests. To integrate with unittest test discovery, include a load_tests() function in your test module:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

There are two main functions for creating unittest.TestSuite instances from text files and modules with doctests:

doctest.DocFileSuite(*paths, [module_relative][, package][, setUp][, tearDown][, globs][, optionflags][, parser][, encoding])

Convert doctest tests from one or more text files to a unittest.TestSuite.

The returned unittest.TestSuite is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a failureException exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Pass one or more paths (as strings) to text files to be examined.

Options may be provided as keyword arguments:

Optional argument module_relative specifies how the filenames in paths should be interpreted:

- If module_relative is True (the default), then each filename in paths specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the package argument is specified, then it is relative to that package. To ensure OS-independence, each filename should use / characters to separate path segments, and may not be an absolute path (i.e., it may not begin with /).
- If module_relative is False, then each filename in paths specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument package is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames in paths. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify package if module_relative is False.

Optional argument setUp specifies a set-up function for the test suite. This is called before running the tests in each file. The setUp function will be passed a DocTest object. The setUp function can access the test globals as the globs attribute of the test passed.

Optional argument tearDown specifies a tear-down function for the test suite. This is called after running the tests in each file. The tearDown function will be passed a DocTest object. The setUp function can access the test globals as the globs attribute of the test passed.

Optional argument globs is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, globs is a new empty dictionary.

Optional argument optionflags specifies the default doctest options for the tests, created by or-ing together individual option flags. See section Option Flags. See function set_unittest_reportflags() below for a better way to set reporting options.

Optional argument parser specifies a DocTestParser (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., DocTestParser()).

Optional argument encoding specifies an encoding that should be used to convert the file to unicode.

New in version 2.4.

Changed in version 2.5: The global __file__ was added to the globals provided to doctests loaded from a text file using DocFileSuite().

Changed in version 2.5: The parameter encoding was added.

Note: Unlike testmod() and DocTestFinder, this function raises a ValueError if module contains no docstrings. You can prevent this error by passing a DocTestFinder instance as the test_finder argument with its exclude_empty keyword argument set to False:

```
>>> finder = doctest.DocTestFinder(exclude_empty=False)
>>> suite = doctest.DocTestSuite(test_finder=finder)
```

---

doctest.DocTestSuite([module][, globs][, extraglobs][, test_finder][, setUp][, tearDown][, checker])
Convert doctest tests for a module to a unittest.TestSuite.

The returned unittest.TestSuite is to be run by the unittest framework and runs each doctest in the module. If any of the doctests fail, then the synthesized unit test fails, and a failureException exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Optional argument module provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not specified, the module calling this function is used.

Optional argument globs is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, globs is a new empty dictionary.

Optional argument extraglobs specifies an extra set of global variables, which is merged into globs. By default, no extra globals are used.

Optional argument test_finder is the DocTestFinder object (or a drop-in replacement) that is used to extract doctests from the module.

Optional arguments setUp, tearDown, and optionflags are the same as for function DocFileSuite() above.

New in version 2.3.

Changed in version 2.4: The parameters globs, extraglobs, test_finder, setUp, tearDown, and optionflags were added; this function now uses the same search technique as testmod().

Under the covers, DocTestSuite() creates a unittest.TestSuite out of doctest.DocTestCase instances, and DocTestCase is a subclass of unittest.TestCase. DocTestCase isn't documented here (it's an internal detail), but studying its code can answer questions about the exact details of unittest integration.

Similarly, DocFileSuite() creates a unittest.TestSuite out of doctest.DocFileCase instances, and DocFileCase is a subclass of DocTestCase.

So both ways of creating a unittest.TestSuite run instances of DocTestCase. This is important for a subtle reason: when you run doctest functions yourself, you can control the doctest options in use directly, by passing option flags to doctest functions. However, if you're writing a unittest framework, unittest ultimately controls when and how tests get run. The framework author typically wants to control doctest reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through unittest to doctest test runners.

For this reason, doctest also supports a notion of doctest reporting flags specific to unittest support, via this function:

doctest.set_unittest_reportflags(flags)
Set the doctest reporting flags to use.

Argument flags or's together option flags. See section Option Flags. Only "reporting flags" can be used.

This is a module-global setting, and affects all future doctests run by module unittest: the runTest() method of DocTestCase looks at the option flags specified for the test case when the DocTestCase instance was constructed. If no reporting flags were specified (which is the typical and expected case), doctest's unittest reporting flags are bitwise ORed into the option flags, and the option flags so augmented are passed to the DocTestRunner instance created to run the doctest. If any reporting flags were specified when the DocTestCase instance was constructed, doctest's unittest reporting flags are ignored.

The value of the unittest reporting flags in effect before the function was called is returned by the function.

New in version 2.4.

25.2.6 Advanced API

The basic API is a simple wrapper that's intended to make doctest easy to use. It is fairly flexible, and should meet most users' needs; however, if you require more fine-grained control over testing, or wish to extend doctest's capabilities, then you should use the advanced API.

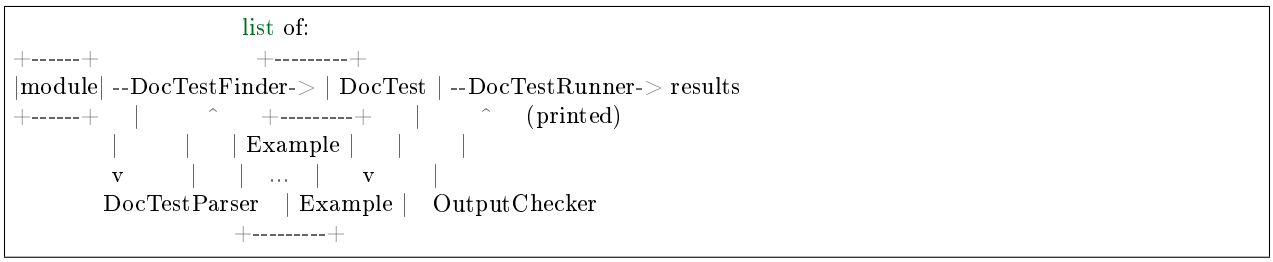
The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- Example: A single Python statement, paired with its expected output.
- DocTest: A collection of Examples, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- DocTestFinder: Finds all docstrings in a given module, and uses a DocTestParser to create a DocTest from every docstring that contains interactive examples.
- DocTestParser: Creates a DocTest object from a string (such as an object's docstring).
- DocTestRunner: Executes the examples in a DocTest, and uses an OutputChecker to verify their output.
- OutputChecker: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:



DocTest Objects

class doctest.DocTest(examples, globs, name, filename, lineno, docstring)

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

New in version 2.4.

DocTest defines the following attributes. They are initialized by the constructor, and should not be modified directly.

examples
: A list of Example objects encoding the individual interactive Python examples that should be run by this test.

globs
: The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in globs after the test is run.

name
: A string name identifying the DocTest. Typically, this is the name of the object or file that the test was extracted from.

filename
: The name of the file that this DocTest was extracted from; or None if the filename is unknown, or if the DocTest was not extracted from a file.

lineno
: The line number within filename where this DocTest begins, or None if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

docstring
: The string that the test was extracted from, or None if the string is unavailable, or if the test was not extracted from a string.

Example Objects

class doctest.Example(source, want[, exc_msg][, lineno][, indent][, options])
: A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

New in version 2.4.

Example defines the following attributes. They are initialized by the constructor, and should not be modified directly.

source
: A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.

want
: The expected output from running the example's source code (either from stdout, or a traceback in case of exception). want ends with a newline unless no output is expected, in which case it's an empty string. The constructor adds a newline when necessary.

exc_msg
: The exception message generated by the example, if the example is expected to generate an exception; or None if it is not expected to generate an exception. This exception message is compared against the return value of traceback.format_exception_only(). exc_msg ends with a newline unless it's None. The constructor adds a newline if needed.

lineno
: The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

indent
: The example's indentation in the containing string, i.e., the number of space characters that precede the example's first prompt.

options
: A dictionary mapping from option flags to True or False, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the DocTestRunner's optionflags). By default, no options are set.

DocTestFinder objects

class doctest.DocTestFinder([verbose][, parser][, recurse][, exclude_empty])
: A processing class used to extract the DocTests that are relevant to a given object, from its docstring and the docstrings of its contained objects. DocTests can currently be extracted from the following object types: modules, functions, classes, methods, staticmethods, classmethods, and properties.

  The optional argument verbose can be used to display the objects searched by the finder. It defaults to False (no output).

  The optional argument parser specifies the DocTestParser object (or a drop-in replacement) that is used to extract doctests from docstrings.

  If the optional argument recurse is false, then DocTestFinder.find() will only examine the given object, and not any contained objects.

  If the optional argument exclude_empty is false, then DocTestFinder.find() will include tests for objects with empty docstrings.

  New in version 2.4.

  DocTestFinder defines the following method:

  find(obj[, name][, module][, globs][, extraglobs])
  : Return a list of the DocTests that are defined by obj's docstring, or by any of its contained objects' docstrings.

    The optional argument name specifies the object's name; this name will be used to construct names for the returned DocTests. If name is not specified, then obj.__name__ is used.

    The optional parameter module is the module that contains the given object. If the module is not specified or is None, then the test finder will attempt to automatically determine the correct module. The object's module is used:

    - As a default namespace, if globs is not specified.
    - To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than module are ignored.)
    - To find the name of the file containing the object.
    - To help find the line number of the object within its file.

    If module is False, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself: if module is False, or is None but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

    The globals for each DocTest is formed by combining globs and extraglobs (bindings in extraglobs override bindings in globs). A new shallow copy of the globals dictionary is created for each DocTest. If globs is not specified, then it defaults to the module's __dict__, if specified, or {} otherwise. If extraglobs is not specified, then it defaults to {}.

DocTestParser objects

class doctest.DocTestParser
A processing class used to extract interactive examples from a string, and use them to create a DocTest object.

New in version 2.4.

DocTestParser defines the following methods:

get_doctest(string, globs, name, filename, lineno)
Extract all doctest examples from the given string, and collect them into a DocTest object.

globs, name, filename, and lineno are attributes for the new DocTest object. See the documentation for DocTest for more information.

get_examples(string[, name])
Extract all doctest examples from the given string, and return them as a list of Example objects. Line numbers are 0-based. The optional argument name is a name identifying this string, and is only used for error messages.

parse(string[, name])
Divide the given string into examples and intervening text, and return them as a list of alternating Examples and strings. Line numbers for the Examples are 0-based. The optional argument name is a name identifying this string, and is only used for error messages.

DocTestRunner objects

class doctest.DocTestRunner([checker][, verbose][, optionflags])
A processing class used to execute and verify the interactive examples in a DocTest.

The comparison between expected outputs and actual outputs is done by an OutputChecker. This comparison may be customized with a number of option flags; see section Option Flags for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of OutputChecker to the constructor.

The test runner's display output can be controlled in two ways. First, an output function can be passed to TestRunner.run(); this function will be called with strings that should be displayed. It defaults to sys.stdout.write. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods report_start(), report_success(), report_unexpected_exception(), and report_failure().

The optional keyword argument checker specifies the OutputChecker object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument verbose controls the DocTestRunner's verbosity. If verbose is True, then information is printed about each example, as it is run. If verbose is False, then only failures are printed. If verbose is unspecified, or None, then verbose output is used iff the command-line switch -v is used.

The optional keyword argument optionflags can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section Option Flags.

New in version 2.4.

DocTestParser defines the following methods:

report_start(out, test, example)
Report that the test runner is about to process the given example. This method is provided to allow subclasses of DocTestRunner to customize their output; it should not be called directly.

example is the example about to be processed. test is the test containing example. out is the output function that was passed to DocTestRunner.run().

report_success(out, test, example, got)
Report that the given example ran successfully. This method is provided to allow subclasses of DocTestRunner to customize their output; it should not be called directly.

example is the example about to be processed. got is the actual output from the example. test is the test containing example. out is the output function that was passed to DocTestRunner.run().

report_failure(out, test, example, got)
Report that the given example failed. This method is provided to allow subclasses of DocTestRunner to customize their output; it should not be called directly.

example is the example about to be processed. got is the actual output from the example. test is the test containing example. out is the output function that was passed to DocTestRunner.run().

report_unexpected_exception(out, test, example, exc_info)
Report that the given example raised an unexpected exception. This method is provided to allow subclasses of DocTestRunner to customize their output; it should not be called directly.

example is the example about to be processed. exc_info is a tuple containing information about the unexpected exception (as returned by sys.exc_info()). test is the test containing example. out is the output function that was passed to DocTestRunner.run().

run(test[, compileflags][, out][, clear_globs])
Run the examples in test (a DocTest object), and display the results using the writer function out.

The examples are run in the namespace test.globs. If clear_globs is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use clear_globs=False.

compileflags gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to globs.

The output of each example is checked using the DocTestRunner's output checker, and the results are formatted by the DocTestRunner.report_*() methods.

summarize([verbose])
Print a summary of all the test cases that have been run by this DocTestRunner, and return a named tuple TestResults(failed, attempted).

The optional verbose argument controls how detailed the summary is. If the verbosity is not specified, then the DocTestRunner's verbosity is used.

Changed in version 2.6: Use a named tuple.

OutputChecker objects

class doctest.OutputChecker
A class used to check the whether the actual output from a doctest example matches the expected output. OutputChecker defines two methods: check_output(), which compares a given pair of outputs, and returns true if they match; and output_difference(), which returns a string describing the differences between two outputs.

New in version 2.4.

OutputChecker defines the following methods:

check_output(want, got, optionflags)
: Return True iff the actual output from an example (got) matches the expected output (want). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section Option Flags for more information about option flags.

output_difference(example, got, optionflags)
: Return a string describing the differences between the expected output for a given example (example) and the actual output (got). optionflags is the set of option flags used to compare want and got.

25.2.7 Debugging

Doctest provides several mechanisms for debugging doctest examples:

- Several functions convert doctests to executable Python programs, which can be run under the Python debugger, pdb.
- The DebugRunner class is a subclass of DocTestRunner that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.
- The unittest cases generated by DocTestSuite() support the debug() method defined by unittest.TestCase.
- You can add a call to pdb.set_trace() in a doctest example, and you'll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose a.py contains just this module docstring:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print x+3
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

Then an interactive Python session may look like this:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print x+3
  3 ->      import pdb; pdb.set_trace()
[EOF]
(Pdb) print x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
```

(continues on next page)

(continued from previous page)

```
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) print x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

Changed in version 2.4: The ability to use pdb.set_trace() usefully inside doctests was added.

Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger:

doctest.script_from_examples(s)

Convert text with examples to a script.

Argument s is a string containing doctest examples. The string is converted to a Python script, where doctest examples in s are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```
import doctest
print doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print x+y
    3
""")
```

displays:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print x+y
# Expected:
## 3
```

This function is used internally by other functions (see below), but can also be useful when you want to transform an interactive Python session into a Python script.

New in version 2.4.

doctest.testsource(module, name)

Convert the doctest for an object to a script.

Argument module is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument name is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as

described for script_from_examples() above. For example, if module a.py contains a top-level function f(), then

```
import a, doctest
print doctest.testsource(a, "a.f")
```

prints a script version of function f()'s docstring, with doctests converted to code, and the rest placed in comments.

New in version 2.3.

doctest.debug(module, name[, pm])
Debug the doctests for an object.

The module and name arguments are the same as for function testsource() above. The synthesized Python script for the named object's docstring is written to a temporary file, and then that file is run under the control of the Python debugger, pdb.

A shallow copy of module.__dict__ is used for both local and global execution context.

Optional argument pm controls whether post-mortem debugging is used. If pm has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If it does, then post-mortem debugging is invoked, via pdb.post_mortem(), passing the traceback object from the unhandled exception. If pm is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate execfile() call to pdb.run().

New in version 2.3.

Changed in version 2.4: The pm argument was added.

doctest.debug_src(src[, pm][, globs])
Debug the doctests in a string.

This is like function debug() above, except that a string containing doctest examples is specified directly, via the src argument.

Optional argument pm has the same meaning as in function debug() above.

Optional argument globs gives a dictionary to use as both local and global execution context. If not specified, or None, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

New in version 2.4.

The DebugRunner class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially DebugRunner's docstring (which is a doctest!) for more details:

class doctest.DebugRunner([checker][, verbose][, optionflags])
A subclass of DocTestRunner that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an UnexpectedException exception is raised, containing the test, the example, and the original exception. If the output doesn't match, then a DocTestFailure exception is raised, containing the test, the example, and the actual output.

For information about the constructor parameters and methods, see the documentation for DocTestRunner in section Advanced API.

There are two exceptions that may be raised by DebugRunner instances:

exception doctest.DocTestFailure(test, example, got)
An exception raised by DocTestRunner to signal that a doctest example's actual output did not match its expected output. The constructor arguments are used to initialize the attributes of the same names.

DocTestFailure defines the following attributes:

DocTestFailure.test
The DocTest object that was being run when the example failed.

DocTestFailure.example
The Example that failed.

DocTestFailure.got
The example's actual output.

exception doctest.UnexpectedException(test, example, exc_info)
An exception raised by DocTestRunner to signal that a doctest example raised an unexpected exception. The constructor arguments are used to initialize the attributes of the same names.

UnexpectedException defines the following attributes:

UnexpectedException.test
The DocTest object that was being run when the example failed.

UnexpectedException.example
The Example that failed.

UnexpectedException.exc_info
A tuple containing information about the unexpected exception, as returned by sys.exc_info().

25.2.8 Soapbox

As mentioned in the introduction, doctest has grown to have three primary uses:

1. Checking examples in docstrings.
2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned—it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I'm still amazed at how often one of my doctest examples stops working after a "harmless" change.

Doctest also makes an excellent tool for regression testing, especially if you don't skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what's actually being tested, and why. When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It's true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that: the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It's a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests:

- Write text files containing test cases as interactive examples, and test the files using testfile() or DocFileSuite(). This is recommended, although is easiest to do for new projects, designed from the start to use doctest.
- Define functions named _regrtest_topic that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.
- Define a __test__ dictionary mapping from regression test topics to docstrings containing test cases.

When you have placed your tests in a module, the module can itself be the test runner. When a test fails, you can arrange for your test runner to re-run only the failing doctest while you debug the problem. Here is a minimal example of such a test runner:

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.REPORT_ONLY_FIRST_FAILURE
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print("{} failures out of {} tests".format(fail, total))
```

25.3 unittest — Unit testing framework

New in version 2.1.

(If you are already familiar with the basic concepts of testing, you might want to skip to the list of assert methods.)

The Python unit testing framework, sometimes referred to as "PyUnit," is a Python language version of JUnit, by Kent Beck and Erich Gamma. JUnit is, in turn, a Java version of Kent's Smalltalk testing framework. Each is the de facto standard unit testing framework for its respective language.

unittest supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. The unittest module provides classes that make it easy to support these qualities for a set of tests.

To achieve this, unittest supports some important concepts:

test fixture A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case A test case is the smallest unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.

test suite A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

The test case and test fixture concepts are supported through the TestCase and FunctionTestCase classes; the former should be used when creating new tests, and the latter can be used when integrating existing test code with a unittest-driven framework. When building test fixtures using TestCase, the setUp() and tearDown() methods can be overridden to provide initialization and cleanup for the fixture. With FunctionTestCase, existing functions can be passed to the constructor for these purposes. When the test is run, the fixture initialization is run first; if it succeeds, the cleanup method is run after the test has been executed, regardless of the outcome of the test. Each instance of the TestCase will only be used to run a single test method, so a new fixture is created for each test.

Test suites are implemented by the TestSuite class. This class allows individual tests and test suites to be aggregated; when the suite is executed, all tests added directly to the suite and in "child" test suites are run.

A test runner is an object that provides a single method, run(), which accepts a TestCase or TestSuite object as a parameter, and returns a result object. The class TestResult is provided for use as the result object. unittest provides the TextTestRunner as an example test runner which reports test results on the standard error stream by default. Alternate runners can be implemented for other environments (such as graphical environments) without any need to derive from a specific class.

See also:

Module doctest Another test-support module with a very different flavor.

unittest2: A backport of new unittest features for Python 2.4-2.6 Many new features were added to unittest in Python 2.7, including test discovery. unittest2 allows you to use these features with earlier versions of Python.

Simple Smalltalk Testing: With Patterns Kent Beck's original paper on testing frameworks using the pattern shared by unittest.

Nose and pytest Third-party unittest frameworks with a lighter-weight syntax for writing tests. For example, assert func(10) == 42.

The Python Testing Tools Taxonomy An extensive list of Python testing tools including functional testing frameworks and mock object libraries.

Testing in Python Mailing List A special-interest-group for discussion of testing, and testing tools, in Python.

25.3.1 Basic example

The unittest module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

Here is a short script to test three string methods:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
```

(continues on next page)

(continued from previous page)

```
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

A testcase is created by subclassing unittest.TestCase. The three individual tests are defined with methods whose names start with the letters test. This naming convention informs the test runner about which methods represent tests.

The crux of each test is a call to assertEqual() to check for an expected result; assertTrue() or assertFalse() to verify a condition; or assertRaises() to verify that a specific exception gets raised. These methods are used instead of the assert statement so the test runner can accumulate all test results and produce a report.

The setUp() and tearDown() methods allow you to define instructions that will be executed before and after each test method. They are covered in more detail in the section Organizing test code.

The final block shows a simple way to run the tests. unittest.main() provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Instead of unittest.main(), there are other ways to run the tests with a finer level of control, less terse output, and no requirement to be run from the command line. For example, the last two lines may be replaced with:

```
suite = unittest.TestLoader().loadTestsFromTestCase(TestStringMethods)
unittest.TextTestRunner(verbosity=2).run(suite)
```

Running the revised script from the interpreter or another script produces the following output:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

The above examples show the most commonly used unittest features which are sufficient to meet many everyday testing needs. The remainder of the documentation explores the full feature set from first principles.

25.3.2 Command-Line Interface

The unittest module can be used from the command line to run tests from modules, classes or even individual test methods:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

You can pass in a list with any combination of module names, and fully qualified class or method names.

You can run tests with more detail (higher verbosity) by passing in the -v flag:

```
python -m unittest -v test_module
```

For a list of all the command-line options:

```
python -m unittest -h
```

Changed in version 2.7: In earlier versions it was only possible to run individual test methods and not modules or classes.

Command-line options

unittest supports these command-line options:

-b, --buffer
: The standard output and standard error streams are buffered during the test run. Output during a passing test is discarded. Output is echoed normally on test fail or error and is added to the failure messages.

-c, --catch
: Control-C during the test run waits for the current test to end and then reports all the results so far. A second Control-C raises the normal KeyboardInterrupt exception.

  See Signal Handling for the functions that provide this functionality.

-f, --failfast
: Stop the test run on the first error or failure.

New in version 2.7: The command-line options -b, -c and -f were added.

The command line can also be used for test discovery, for running all of the tests in a project or just a subset.

25.3.3 Test Discovery

New in version 2.7.

Unittest supports simple test discovery. In order to be compatible with test discovery, all of the test files must be modules or packages importable from the top-level directory of the project (this means that their filenames must be valid identifiers).

Test discovery is implemented in TestLoader.discover(), but can also be used from the command line. The basic command-line usage is:

```
cd project_directory
python -m unittest discover
```

The discover sub-command has the following options:

-v, --verbose
: Verbose output

-s, --start-directory directory
: Directory to start discovery (. default)

-p, --pattern pattern
: Pattern to match test files (test*.py default)

-t, --top-level-directory directory
    Top level directory of project (defaults to start directory)

The -s, -p, and -t options can be passed in as positional arguments in that order. The following two command lines are equivalent:

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

As well as being a path it is possible to pass a package name, for example myproject.subpackage.test, as the start directory. The package name you supply will then be imported and its location on the filesystem will be used as the start directory.

> Caution: Test discovery loads tests by importing them. Once test discovery has found all the test files from the start directory you specify it turns the paths into package names to import. For example foo/bar/baz.py will be imported as foo.bar.baz.
>
> If you have a package installed globally and attempt test discovery on a different copy of the package then the import could happen from the wrong place. If this happens test discovery will warn you and exit.
>
> If you supply the start directory as a package name rather than a path to a directory then discover assumes that whichever location it imports from is the location you intended, so you will not get the warning.

Test modules and packages can customize test loading and discovery by through the load_tests protocol.

25.3.4 Organizing test code

The basic building blocks of unit testing are test cases — single scenarios that must be set up and checked for correctness. In unittest, test cases are represented by instances of unittest's TestCase class. To make your own test cases you must write subclasses of TestCase, or use FunctionTestCase.

An instance of a TestCase-derived class is an object that can completely run a single test method, together with optional set-up and tidy-up code.

The testing code of a TestCase instance should be entirely self contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.

The simplest TestCase subclass will simply override the runTest() method in order to perform specific testing code:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50), 'incorrect default size')
```

Note that in order to test something, we use one of the assert*() methods provided by the TestCase base class. If the test fails, an exception will be raised, and unittest will identify the test case as a failure. Any other exceptions will be treated as errors. This helps you identify where the problem is: failures are caused by incorrect results - a 5 where you expected a 6. Errors are caused by incorrect code - e.g., a TypeError caused by an incorrect function call.

The way to run a test case will be described later. For now, note that to construct an instance of such a test case, we call its constructor without arguments:

```
testCase = DefaultWidgetSizeTestCase()
```

Now, such test cases can be numerous, and their set-up can be repetitive. In the above case, constructing a Widget in each of 100 Widget test case subclasses would mean unsightly duplication.

Luckily, we can factor out such set-up code by implementing a method called setUp(), which the testing framework will automatically call for us when we run the test:

```
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

class DefaultWidgetSizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

class WidgetResizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

If the setUp() method raises an exception while the test is running, the framework will consider the test to have suffered an error, and the runTest() method will not be executed.

Similarly, we can provide a tearDown() method that tidies up after the runTest() method has been run:

```
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
        self.widget = None
```

If setUp() succeeded, the tearDown() method will be run whether runTest() succeeded or not.

Such a working environment for the testing code is called a fixture.

Often, many small test cases will use the same fixture. In this case, we would end up subclassing SimpleWidgetTestCase into many small one-method classes such as DefaultWidgetSizeTestCase. This is time-consuming and discouraging, so in the same vein as JUnit, unittest provides a simpler mechanism:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

(continues on next page)

(continued from previous page)

```
        self.widget = None

    def test_default_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

Here we have not provided a runTest() method, but have instead provided two different test methods. Class instances will now each run one of the test_*() methods, with self.widget created and destroyed separately for each instance. When creating an instance we must specify the test method it is to run. We do this by passing the method name in the constructor:

```
defaultSizeTestCase = WidgetTestCase('test_default_size')
resizeTestCase = WidgetTestCase('test_resize')
```

Test case instances are grouped together according to the features they test. unittest provides a mechanism for this: the test suite, represented by unittest's TestSuite class:

```
widgetTestSuite = unittest.TestSuite()
widgetTestSuite.addTest(WidgetTestCase('test_default_size'))
widgetTestSuite.addTest(WidgetTestCase('test_resize'))
```

For the ease of running tests, as we will see later, it is a good idea to provide in each test module a callable object that returns a pre-built test suite:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_size'))
    suite.addTest(WidgetTestCase('test_resize'))
    return suite
```

or even:

```
def suite():
    tests = ['test_default_size', 'test_resize']

    return unittest.TestSuite(map(WidgetTestCase, tests))
```

Since it is a common pattern to create a TestCase subclass with many similarly named test functions, unittest provides a TestLoader class that can be used to automate the process of creating a test suite and populating it with individual tests. For example,

```
suite = unittest.TestLoader().loadTestsFromTestCase(WidgetTestCase)
```

will create a test suite that will run WidgetTestCase.test_default_size() and WidgetTestCase.test_resize. TestLoader uses the 'test' method name prefix to identify test methods automatically.

Note that the order in which the various test cases will be run is determined by sorting the test function names with respect to the built-in ordering for strings.

Often it is desirable to group suites of test cases together, so as to run tests for the whole system at once. This is easy, since TestSuite instances can be added to a TestSuite just as TestCase instances can be added to a TestSuite:

```
suite1 = module1.TheTestSuite()
suite2 = module2.TheTestSuite()
alltests = unittest.TestSuite([suite1, suite2])
```

You can place the definitions of test cases and test suites in the same modules as the code they are to test (such as widget.py), but there are several advantages to placing the test code in a separate module, such as test_widget.py:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
- There is less temptation to change test code to fit the code it tests without a good reason.
- Test code should be modified much less frequently than the code it tests.
- Tested code can be refactored more easily.
- Tests for modules written in C must be in separate modules anyway, so why not be consistent?
- If the testing strategy changes, there is no need to change the source code.

25.3.5 Re-using old test code

Some users will find that they have existing test code that they would like to run from unittest, without converting every old test function to a TestCase subclass.

For this reason, unittest provides a FunctionTestCase class. This subclass of TestCase can be used to wrap an existing test function. Set-up and tear-down functions can also be provided.

Given the following test function:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows:

```
testcase = unittest.FunctionTestCase(testSomething)
```

If there are additional set-up and tear-down methods that should be called as part of the test case's operation, they can also be provided like so:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

To make migrating existing test suites easier, unittest supports tests raising AssertionError to indicate test failure. However, it is recommended that you use the explicit TestCase.fail*() and TestCase.assert*() methods instead, as future versions of unittest may treat AssertionError differently.

Note: Even though FunctionTestCase can be used to quickly convert an existing test base over to a unittest-based system, this approach is not recommended. Taking the time to set up proper TestCase subclasses will make future test refactorings infinitely easier.

In some cases, the existing tests may have been written using the doctest module. If so, doctest provides a DocTestSuite class that can automatically build unittest.TestSuite instances from the existing doctest-based tests.

25.3.6 Skipping tests and expected failures

New in version 2.7.

Unittest supports skipping individual test methods and even whole classes of tests. In addition, it supports marking a test as an "expected failure," a test that is broken and will fail, but shouldn't be counted as a failure on a TestResult.

Skipping a test is simply a matter of using the skip() decorator or one of its conditional variants.

Basic skipping looks like this:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

This is the output of running the example above in verbose mode:

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this library version'
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating skipping'
test_windows_support (__main__.MyTestCase) ... skipped 'requires Windows'

----------------------------------------------------------------------
Ran 3 tests in 0.005s

OK (skipped=3)
```

Classes can be skipped just like methods:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp() can also skip the test. This is useful when a resource that needs to be set up is not available.

Expected failures use the expectedFailure() decorator.

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

It's easy to roll your own skipping decorators by making a decorator that calls skip() on the test when it wants it to be skipped. This decorator skips the test unless the passed object has a certain attribute:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

The following decorators implement test skipping and expected failures:

unittest.skip(reason)
: Unconditionally skip the decorated test. reason should describe why the test is being skipped.

unittest.skipIf(condition, reason)
: Skip the decorated test if condition is true.

unittest.skipUnless(condition, reason)
: Skip the decorated test unless condition is true.

unittest.expectedFailure()
: Mark the test as an expected failure. If the test fails when run, the test is not counted as a failure.

exception unittest.SkipTest(reason)
: This exception is raised to skip a test.

  Usually you can use TestCase.skipTest() or one of the skipping decorators instead of raising this directly.

Skipped tests will not have setUp() or tearDown() run around them. Skipped classes will not have setUpClass() or tearDownClass() run.

25.3.7 Classes and functions

This section describes in depth the API of unittest.

Test cases

class unittest.TestCase(methodName='runTest')
: Instances of the TestCase class represent the smallest testable units in the unittest universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the test, and methods that the test code can use to check for and report various kinds of failure.

  Each instance of TestCase will run a single test method: the method named methodName. If you remember, we had an earlier example that went something like this:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_size'))
    suite.addTest(WidgetTestCase('test_resize'))
    return suite
```

Here, we create two instances of WidgetTestCase, each of which runs a single test.

methodName defaults to runTest().

TestCase instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

Methods in the first group (running the test) are:

setUp()
: Method called to prepare the test fixture. This is called immediately before calling the test method; other than AssertionError or SkipTest, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

tearDown()
: Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than AssertionError or SkipTest, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the setUp() succeeds, regardless of the outcome of the test method. The default implementation does nothing.

setUpClass()
: A class method called before tests in an individual class are run. setUpClass is called with the class as the only argument and must be decorated as a classmethod():

```
@classmethod
def setUpClass(cls):
    ...
```

See Class and Module Fixtures for more details.

New in version 2.7.

tearDownClass()
: A class method called after tests in an individual class have run. tearDownClass is called with the class as the only argument and must be decorated as a classmethod():

```
@classmethod
def tearDownClass(cls):
    ...
```

See Class and Module Fixtures for more details.

New in version 2.7.

run(result=None)
: Run the test, collecting the result into the test result object passed as result. If result is omitted or None, a temporary result object is created (by calling the defaultTestResult() method) and used. The result object is not returned to run()'s caller.

The same effect may be had by simply calling the TestCase instance.

skipTest(reason)
: Calling this during a test method or setUp() skips the current test. See Skipping tests and expected failures for more information.

New in version 2.7.

debug()
Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger.

The TestCase class provides several assert methods to check for and report failures. The following table lists the most commonly used methods (see the tables below for more assert methods):

Method	Checks that	New in
assertEqual(a, b)	a == b	
assertNotEqual(a, b)	a != b	
assertTrue(x)	bool(x) is True	
assertFalse(x)	bool(x) is False	
assertIs(a, b)	a is b	2.7
assertIsNot(a, b)	a is not b	2.7
assertIsNone(x)	x is None	2.7
assertIsNotNone(x)	x is not None	2.7
assertIn(a, b)	a in b	2.7
assertNotIn(a, b)	a not in b	2.7
assertIsInstance(a, b)	isinstance(a, b)	2.7
assertNotIsInstance(a, b)	not isinstance(a, b)	2.7

All the assert methods (except assertRaises(), assertRaisesRegexp()) accept a msg argument that, if specified, is used as the error message on failure (see also longMessage).

assertEqual(first, second, msg=None)
Test that first and second are equal. If the values do not compare equal, the test will fail.

In addition, if first and second are the exact same type and one of list, tuple, dict, set, frozenset or unicode or any type that a subclass registers with addTypeEqualityFunc() the type-specific equality function will be called in order to generate a more useful default error message (see also the list of type-specific methods).

Changed in version 2.7: Added the automatic calling of type-specific equality function.

assertNotEqual(first, second, msg=None)
Test that first and second are not equal. If the values do compare equal, the test will fail.

assertTrue(expr, msg=None)
assertFalse(expr, msg=None)
Test that expr is true (or false).

Note that this is equivalent to bool(expr) is True and not to expr is True (use assertIs(expr, True) for the latter). This method should also be avoided when more specific methods are available (e.g. assertEqual(a, b) instead of assertTrue(a == b)), because they provide a better error message in case of failure.

assertIs(first, second, msg=None)
assertIsNot(first, second, msg=None)
Test that first and second evaluate (or don't evaluate) to the same object.

New in version 2.7.

assertIsNone(expr, msg=None)
assertIsNotNone(expr, msg=None)
Test that expr is (or is not) None.

New in version 2.7.

assertIn(first, second, msg=None)

assertNotIn(first, second, msg=None)
Test that first is (or is not) in second.

New in version 2.7.

assertIsInstance(obj, cls, msg=None)
assertNotIsInstance(obj, cls, msg=None)
Test that obj is (or is not) an instance of cls (which can be a class or a tuple of classes, as supported by isinstance()). To check for the exact type, use assertIs(type(obj), cls).

New in version 2.7.

It is also possible to check that exceptions and warnings are raised using the following methods:

Method	Checks that	New in
assertRaises(exc, fun, *args, **kwds)	fun(*args, **kwds) raises exc	
assertRaisesRegexp(exc, r, fun, *args, **kwds)	fun(*args, **kwds) raises exc and the message matches regex r	2.7

assertRaises(exception, callable, *args, **kwds)
assertRaises(exception)
Test that an exception is raised when callable is called with any positional or keyword arguments that are also passed to assertRaises(). The test passes if exception is raised, is an error if another exception is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as exception.

If only the exception argument is given, returns a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertRaises(SomeException):
    do_something()
```

The context manager will store the caught exception object in its exception attribute. This can be useful if the intention is to perform additional checks on the exception raised:

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

Changed in version 2.7: Added the ability to use assertRaises() as a context manager.

assertRaisesRegexp(exception, regexp, callable, *args, **kwds)
assertRaisesRegexp(exception, regexp)
Like assertRaises() but also tests that regexp matches on the string representation of the raised exception. regexp may be a regular expression object or a string containing a regular expression suitable for use by re.search(). Examples:

```
self.assertRaisesRegexp(ValueError, "invalid literal for.*XYZ'$",
                        int, 'XYZ')
```

or:

```
with self.assertRaisesRegexp(ValueError, 'literal'):
  int('XYZ')
```

New in version 2.7.

There are also other methods used to perform more specific checks, such as:

Method	Checks that	New in
assertAlmostEqual(a, b)	round(a-b, 7) == 0	
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0	
assertGreater(a, b)	a > b	2.7
assertGreaterEqual(a, b)	a >= b	2.7
assertLess(a, b)	a < b	2.7
assertLessEqual(a, b)	a <= b	2.7
assertRegexpMatches(s, r)	r.search(s)	2.7
assertNotRegexpMatches(s, r)	not r.search(s)	2.7
assertItemsEqual(a, b)	sorted(a) == sorted(b) and works with unhashable objs	2.7
assertDictContainsSubset(a, b)	all the key/value pairs in a exist in b	2.7

assertAlmostEqual(first, second, places=7, msg=None, delta=None)
assertNotAlmostEqual(first, second, places=7, msg=None, delta=None)
Test that first and second are approximately (or not approximately) equal by computing the difference, rounding to the given number of decimal places (default 7), and comparing to zero. Note that these methods round the values to the given number of decimal places (i.e. like the round() function) and not significant digits.

If delta is supplied instead of places then the difference between first and second must be less or equal to (or greater than) delta.

Supplying both delta and places raises a TypeError.

Changed in version 2.7: assertAlmostEqual() automatically considers almost equal objects that compare equal. assertNotAlmostEqual() automatically fails if the objects compare equal. Added the delta keyword argument.

assertGreater(first, second, msg=None)
assertGreaterEqual(first, second, msg=None)
assertLess(first, second, msg=None)
assertLessEqual(first, second, msg=None)
Test that first is respectively >, >=, < or <= than second depending on the method name. If not, the test will fail:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

New in version 2.7.

assertRegexpMatches(text, regexp, msg=None)
Test that a regexp search matches text. In case of failure, the error message will include the pattern and the text (or the pattern and the part of text that unexpectedly matched). regexp may be a regular expression object or a string containing a regular expression suitable for use by re.search().

New in version 2.7.

assertNotRegexpMatches(text, regexp, msg=None)
Verifies that a regexp search does not match text. Fails with an error message including the pattern and the part of text that matches. regexp may be a regular expression object or a string containing a regular expression suitable for use by re.search().

New in version 2.7.

assertItemsEqual(actual, expected, msg=None)
Test that sequence expected contains the same elements as actual, regardless of their order. When they don't, an error message listing the differences between the sequences will be generated.

Duplicate elements are not ignored when comparing actual and expected. It verifies if each element has the same count in both sequences. It is the equivalent of assertEqual(sorted(expected), sorted(actual)) but it works with sequences of unhashable objects as well.

In Python 3, this method is named assertCountEqual.

New in version 2.7.

assertDictContainsSubset(expected, actual, msg=None)
Tests whether the key/value pairs in dictionary actual are a superset of those in expected. If not, an error message listing the missing keys and mismatched values is generated.

New in version 2.7.

Deprecated since version 3.2.

The assertEqual() method dispatches the equality check for objects of the same type to different type-specific methods. These methods are already implemented for most of the built-in types, but it's also possible to register new methods using addTypeEqualityFunc():

addTypeEqualityFunc(typeobj, function)
Registers a type-specific method called by assertEqual() to check if two objects of exactly the same typeobj (not subclasses) compare equal. function must take two positional arguments and a third msg=None keyword argument just as assertEqual() does. It must raise self.failureException(msg) when inequality between the first two parameters is detected – possibly providing useful information and explaining the inequalities in details in the error message.

New in version 2.7.

The list of type-specific methods automatically used by assertEqual() are summarized in the following table. Note that it's usually not necessary to invoke these methods directly.

Method	Used to compare	New in
assertMultiLineEqual(a, b)	strings	2.7
assertSequenceEqual(a, b)	sequences	2.7
assertListEqual(a, b)	lists	2.7
assertTupleEqual(a, b)	tuples	2.7
assertSetEqual(a, b)	sets or frozensets	2.7
assertDictEqual(a, b)	dicts	2.7

assertMultiLineEqual(first, second, msg=None)
Test that the multiline string first is equal to the string second. When not equal a diff of the two strings highlighting the differences will be included in the error message. This method is used by default when comparing Unicode strings with assertEqual().

New in version 2.7.

assertSequenceEqual(seq1, seq2, msg=None, seq_type=None)
Tests that two sequences are equal. If a seq_type is supplied, both seq1 and seq2 must be instances of seq_type or a failure will be raised. If the sequences are different an error message is constructed that shows the difference between the two.

This method is not called directly by assertEqual(), but it's used to implement assertListEqual() and assertTupleEqual().

New in version 2.7.

assertListEqual(list1, list2, msg=None)
assertTupleEqual(tuple1, tuple2, msg=None)
Tests that two lists or tuples are equal. If not, an error message is constructed that shows only the differences between the two. An error is also raised if either of the parameters are of the wrong type. These methods are used by default when comparing lists or tuples with assertEqual().

New in version 2.7.

assertSetEqual(set1, set2, msg=None)
Tests that two sets are equal. If not, an error message is constructed that lists the differences between the sets. This method is used by default when comparing sets or frozensets with assertEqual().

Fails if either of set1 or set2 does not have a set.difference() method.

New in version 2.7.

assertDictEqual(expected, actual, msg=None)
Test that two dictionaries are equal. If not, an error message is constructed that shows the differences in the dictionaries. This method will be used by default to compare dictionaries in calls to assertEqual().

New in version 2.7.

Finally the TestCase provides the following methods and attributes:

fail(msg=None)
Signals a test failure unconditionally, with msg or None for the error message.

failureException
This class attribute gives the exception raised by the test method. If a test framework needs to use a specialized exception, possibly to carry additional information, it must subclass this exception in order to "play fair" with the framework. The initial value of this attribute is AssertionError.

longMessage
If set to True then any explicit failure message you pass in to the assert methods will be appended to the end of the normal failure message. The normal messages contain useful information about the objects involved, for example the message from assertEqual shows you the repr of the two unequal objects. Setting this attribute to True allows you to have a custom error message in addition to the normal one.

This attribute defaults to False, meaning that a custom message passed to an assert method will silence the normal message.

The class setting can be overridden in individual tests by assigning an instance attribute to True or False before calling the assert methods.

New in version 2.7.

maxDiff
This attribute controls the maximum length of diffs output by assert methods that report diffs on failure. It defaults to 80*8 characters. Assert methods affected by this attribute are assertSequenceEqual() (including all the sequence comparison methods that delegate to it), assertDictEqual() and assertMultiLineEqual().

Setting maxDiff to None means that there is no maximum length of diffs.

New in version 2.7.

Testing frameworks can use the following methods to collect information on the test:

countTestCases()
: Return the number of tests represented by this test object. For TestCase instances, this will always be 1.

defaultTestResult()
: Return an instance of the test result class that should be used for this test case class (if no other result instance is provided to the run() method).

  For TestCase instances, this will always be an instance of TestResult; subclasses of TestCase should override this as necessary.

id()
: Return a string identifying the specific test case. This is usually the full name of the test method, including the module and class name.

shortDescription()
: Returns a description of the test, or None if no description has been provided. The default implementation of this method returns the first line of the test method's docstring, if available, or None.

addCleanup(function, *args, **kwargs)
: Add a function to be called after tearDown() to cleanup resources used during the test. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into addCleanup() when they are added.

  If setUp() fails, meaning that tearDown() is not called, then any cleanup functions added will still be called.

  New in version 2.7.

doCleanups()
: This method is called unconditionally after tearDown(), or after setUp() if setUp() raises an exception.

  It is responsible for calling all the cleanup functions added by addCleanup(). If you need cleanup functions to be called prior to tearDown() then you can call doCleanups() yourself.

  doCleanups() pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

  New in version 2.7.

class unittest.FunctionTestCase(testFunc, setUp=None, tearDown=None, description=None)
: This class implements the portion of the TestCase interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a unittest-based test framework.

Deprecated aliases

For historical reasons, some of the TestCase methods had one or more aliases that are now deprecated. The following table lists the correct names along with their deprecated aliases:

Method Name	Deprecated alias(es)
assertEqual()	failUnlessEqual, assertEquals
assertNotEqual()	failIfEqual
assertTrue()	failUnless, assert_
assertFalse()	failIf
assertRaises()	failUnlessRaises
assertAlmostEqual()	failUnlessAlmostEqual
assertNotAlmostEqual()	failIfAlmostEqual

Deprecated since version 2.7: the aliases listed in the second column

Grouping tests

class unittest.TestSuite(tests=())

This class represents an aggregation of individual test cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case. Running a TestSuite instance is the same as iterating over the suite, running each test individually.

If tests is given, it must be an iterable of individual test cases or other test suites that will be used to build the suite initially. Additional methods are provided to add test cases and suites to the collection later on.

TestSuite objects behave much like TestCase objects, except they do not actually implement a test. Instead, they are used to aggregate tests into groups of tests that should be run together. Some additional methods are available to add tests to TestSuite instances:

addTest(test)

Add a TestCase or TestSuite to the suite.

addTests(tests)

Add all the tests from an iterable of TestCase and TestSuite instances to this test suite.

This is equivalent to iterating over tests, calling addTest() for each element.

TestSuite shares the following methods with TestCase:

run(result)

Run the tests associated with this suite, collecting the result into the test result object passed as result. Note that unlike TestCase.run(), TestSuite.run() requires the result object to be passed in.

debug()

Run the tests associated with this suite without collecting the result. This allows exceptions raised by the test to be propagated to the caller and can be used to support running tests under a debugger.

countTestCases()

Return the number of tests represented by this test object, including all individual tests and sub-suites.

__iter__()

Tests grouped by a TestSuite are always accessed by iteration. Subclasses can lazily provide tests by overriding __iter__(). Note that this method maybe called several times on a single suite (for example when counting tests or comparing for equality) so the tests returned must be the same for repeated iterations.

Changed in version 2.7: In earlier versions the TestSuite accessed tests directly rather than through iteration, so overriding __iter__() wasn't sufficient for providing tests.

In the typical usage of a TestSuite object, the run() method is invoked by a TestRunner rather than by the end-user test harness.

Loading and running tests

class unittest.TestLoader

The TestLoader class is used to create test suites from classes and modules. Normally, there is no need to create an instance of this class; the unittest module provides an instance that can be shared as unittest.defaultTestLoader. Using a subclass or instance, however, allows customization of some configurable properties.

TestLoader objects have the following methods:

loadTestsFromTestCase(testCaseClass)

Return a suite of all test cases contained in the TestCase-derived testCaseClass.

loadTestsFromModule(module)

Return a suite of all test cases contained in the given module. This method searches module for classes derived from TestCase and creates an instance of the class for each test method defined for the class.

Note: While using a hierarchy of TestCase-derived classes can be convenient in sharing fixtures and helper functions, defining test methods on base classes that are not intended to be instantiated directly does not play well with this method. Doing so, however, can be useful when the fixtures are different and defined in subclasses.

If a module provides a load_tests function it will be called to load the tests. This allows modules to customize test loading. This is the load_tests protocol.

Changed in version 2.7: Support for load_tests added.

loadTestsFromName(name, module=None)

Return a suite of all test cases given a string specifier.

The specifier name is a "dotted name" that may resolve either to a module, a test case class, a test method within a test case class, a TestSuite instance, or a callable object which returns a TestCase or TestSuite instance. These checks are applied in the order listed here; that is, a method on a possible test case class will be picked up as "a test method within a test case class", rather than "a callable object".

For example, if you have a module SampleTests containing a TestCase-derived class SampleTestCase with three test methods (test_one(), test_two(), and test_three()), the specifier 'SampleTests.SampleTestCase' would cause this method to return a suite which will run all three test methods. Using the specifier 'SampleTests.SampleTestCase.test_two' would cause it to return a test suite which will run only the test_two() test method. The specifier can refer to modules and packages which have not been imported; they will be imported as a side-effect.

The method optionally resolves name relative to the given module.

loadTestsFromNames(names, module=None)

Similar to loadTestsFromName(), but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

getTestCaseNames(testCaseClass)

Return a sorted sequence of method names found within testCaseClass; this should be a subclass of TestCase.

discover(start_dir, pattern='test*.py', top_level_dir=None)
Find all the test modules by recursing into subdirectories from the specified start directory, and return a TestSuite object containing them. Only test files that match pattern will be loaded. (Using shell style pattern matching.) Only module names that are importable (i.e. are valid Python identifiers) will be loaded.

All test modules must be importable from the top level of the project. If the start directory is not the top level directory then the top level directory must be specified separately.

If importing a module fails, for example due to a syntax error, then this will be recorded as a single error and discovery will continue.

If a test package name (directory with __init__.py) matches the pattern then the package will be checked for a load_tests function. If this exists then it will be called with loader, tests, pattern.

If load_tests exists then discovery does not recurse into the package, load_tests is responsible for loading all tests in the package.

The pattern is deliberately not stored as a loader attribute so that packages can continue discovery themselves. top_level_dir is stored so load_tests does not need to pass this argument in to loader.discover().

start_dir can be a dotted module name as well as a directory.

New in version 2.7.

The following attributes of a TestLoader can be configured either by subclassing or assignment on an instance:

testMethodPrefix
String giving the prefix of method names which will be interpreted as test methods. The default value is 'test'.

This affects getTestCaseNames() and all the loadTestsFrom*() methods.

sortTestMethodsUsing
Function to be used to compare method names when sorting them in getTestCaseNames() and all the loadTestsFrom*() methods. The default value is the built-in cmp() function; the attribute can also be set to None to disable the sort.

suiteClass
Callable object that constructs a test suite from a list of tests. No methods on the resulting object are needed. The default value is the TestSuite class.

This affects all the loadTestsFrom*() methods.

class unittest.TestResult
This class is used to compile information about which tests have succeeded and which have failed.

A TestResult object stores the results of a set of tests. The TestCase and TestSuite classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

Testing frameworks built on top of unittest may want access to the TestResult object generated by running a set of tests for reporting purposes; a TestResult instance is returned by the TestRunner.run() method for this purpose.

TestResult instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

errors
A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception.

Changed in version 2.2: Contains formatted tracebacks instead of sys.exc_info() results.

failures
A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the TestCase.assert*() methods.

Changed in version 2.2: Contains formatted tracebacks instead of sys.exc_info() results.

skipped
A list containing 2-tuples of TestCase instances and strings holding the reason for skipping the test.

New in version 2.7.

expectedFailures
A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents an expected failure of the test case.

unexpectedSuccesses
A list containing TestCase instances that were marked as expected failures, but succeeded.

shouldStop
Set to True when the execution of tests should stop by stop().

testsRun
The total number of tests run so far.

buffer
If set to true, sys.stdout and sys.stderr will be buffered in between startTest() and stopTest() being called. Collected output will only be echoed onto the real sys.stdout and sys.stderr if the test fails or errors. Any output is also attached to the failure / error message.

New in version 2.7.

failfast
If set to true stop() will be called on the first failure or error, halting the test run.

New in version 2.7.

wasSuccessful()
Return True if all tests run so far have passed, otherwise returns False.

stop()
This method can be called to signal that the set of tests being run should be aborted by setting the shouldStop attribute to True. TestRunner objects should respect this flag and return without running any additional tests.

For example, this feature is used by the TextTestRunner class to stop the test framework when the user signals an interrupt from the keyboard. Interactive tools which provide TestRunner implementations can use this in a similar manner.

The following methods of the TestResult class are used to maintain the internal data structures, and may be extended in subclasses to support additional reporting requirements. This is particularly useful in building tools which support interactive reporting while tests are being run.

startTest(test)
Called when the test case test is about to be run.

stopTest(test)
Called after the test case test has been executed, regardless of the outcome.

startTestRun()
Called once before any tests are executed.

New in version 2.7.

stopTestRun()
Called once after all tests are executed.

New in version 2.7.

addError(test, err)
Called when the test case test raises an unexpected exception. err is a tuple of the form returned by sys.exc_info(): (type, value, traceback).

The default implementation appends a tuple (test, formatted_err) to the instance's errors attribute, where formatted_err is a formatted traceback derived from err.

addFailure(test, err)
Called when the test case test signals a failure. err is a tuple of the form returned by sys.exc_info(): (type, value, traceback).

The default implementation appends a tuple (test, formatted_err) to the instance's failures attribute, where formatted_err is a formatted traceback derived from err.

addSuccess(test)
Called when the test case test succeeds.

The default implementation does nothing.

addSkip(test, reason)
Called when the test case test is skipped. reason is the reason the test gave for skipping.

The default implementation appends a tuple (test, reason) to the instance's skipped attribute.

addExpectedFailure(test, err)
Called when the test case test fails, but was marked with the expectedFailure() decorator.

The default implementation appends a tuple (test, formatted_err) to the instance's expectedFailures attribute, where formatted_err is a formatted traceback derived from err.

addUnexpectedSuccess(test)
Called when the test case test was marked with the expectedFailure() decorator, but succeeded.

The default implementation appends the test to the instance's unexpectedSuccesses attribute.

class unittest.TextTestResult(stream, descriptions, verbosity)
A concrete implementation of TestResult used by the TextTestRunner.

New in version 2.7: This class was previously named _TextTestResult. The old name still exists as an alias but is deprecated.

unittest.defaultTestLoader
Instance of the TestLoader class intended to be shared. If no customization of the TestLoader is needed, this instance can be used instead of repeatedly creating new instances.

class unittest.TextTestRunner(stream=sys.stderr, descriptions=True, verbosity=1, failfast=False, buffer=False, resultclass=None)
A basic test runner implementation which prints results on standard error. It has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations.

_makeResult()
This method returns the instance of TestResult used by run(). It is not intended to be called directly, but can be overridden in subclasses to provide a custom TestResult.

_makeResult() instantiates the class or callable passed in the TextTestRunner constructor as the resultclass argument. It defaults to TextTestResult if no resultclass is provided. The result class is instantiated with the following arguments:

```
stream, descriptions, verbosity
```

unittest.main([module[, defaultTest[, argv[, testRunner[, testLoader[, exit[, verbosity[, failfast[, catchbreak[, buffer]]]]]]]]]])

A command-line program that loads a set of tests from module and runs them; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

```
if __name__ == '__main__':
    unittest.main()
```

You can run tests with more detailed information by passing in the verbosity argument:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

The defaultTest argument is the name of the test to run if no test names are specified via argv. If not specified or None and no test names are provided via argv, all tests found in module are run.

The argv argument can be a list of options passed to the program, with the first element being the program name. If not specified or None, the values of sys.argv are used.

The testRunner argument can either be a test runner class or an already created instance of it. By default main calls sys.exit() with an exit code indicating success or failure of the tests run.

The testLoader argument has to be a TestLoader instance, and defaults to defaultTestLoader.

main supports being used from the interactive interpreter by passing in the argument exit=False. This displays the result on standard output without calling sys.exit():

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

The failfast, catchbreak and buffer parameters have the same effect as the same-name command-line options.

Calling main actually returns an instance of the TestProgram class. This stores the result of the tests run as the result attribute.

Changed in version 2.7: The exit, verbosity, failfast, catchbreak and buffer parameters were added.

load_tests Protocol

New in version 2.7.

Modules or packages can customize how tests are loaded from them during normal test runs or test discovery by implementing a function called load_tests.

If a test module defines load_tests it will be called by TestLoader.loadTestsFromModule() with the following arguments:

```
load_tests(loader, standard_tests, None)
```

It should return a TestSuite.

loader is the instance of TestLoader doing the loading. standard_tests are the tests that would be loaded by default from the module. It is common for test modules to only want to add or remove tests from the standard set of tests. The third argument is used when loading packages as part of test discovery.

A typical load_tests function that loads tests from a specific set of TestCase classes may look like:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

If discovery is started, either from the command line or by calling TestLoader.discover(), with a pattern that matches a package name then the package __init__.py will be checked for load_tests.

Note: The default pattern is 'test*.py'. This matches all Python files that start with 'test' but won't match any test directories.

A pattern like 'test*' will match test packages as well as modules.

If the package __init__.py defines load_tests then it will be called and discovery not continued into the package. load_tests is called with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

This should return a TestSuite representing all the tests from the package. (standard_tests will only contain tests collected from __init__.py.)

Because the pattern is passed into load_tests the package is free to continue (and potentially modify) test discovery. A 'do nothing' load_tests function for a test package would look like:

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

25.3.8 Class and Module Fixtures

Class and module level fixtures are implemented in TestSuite. When the test suite encounters a test from a new class then tearDownClass() from the previous class (if there is one) is called, followed by setUpClass() from the new class.

Similarly if a test is from a different module from the previous test then tearDownModule from the previous module is run, followed by setUpModule from the new module.

After all the tests have run the final tearDownClass and tearDownModule are run.

Note that shared fixtures do not play well with [potential] features like test parallelization and they break test isolation. They should be used with care.

The default ordering of tests created by the unittest test loaders is to group all tests from the same modules and classes together. This will lead to setUpClass / setUpModule (etc) being called exactly once per class and module. If you randomize the order, so that tests from different modules and classes are adjacent to each other, then these shared fixture functions may be called multiple times in a single test run.

Shared fixtures are not intended to work with suites with non-standard ordering. A BaseTestSuite still exists for frameworks that don't want to support shared fixtures.

If there are any exceptions raised during one of the shared fixture functions the test is reported as an error. Because there is no corresponding test instance an _ErrorHolder object (that has the same interface as a TestCase) is created to represent the error. If you are just using the standard unittest test runner then this detail doesn't matter, but if you are a framework author it may be relevant.

setUpClass and tearDownClass

These must be implemented as class methods:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

If you want the setUpClass and tearDownClass on base classes called then you must call up to them yourself. The implementations in TestCase are empty.

If an exception is raised during a setUpClass then the tests in the class are not run and the tearDownClass is not run. Skipped classes will not have setUpClass or tearDownClass run. If the exception is a SkipTest exception then the class will be reported as having been skipped instead of as an error.

setUpModule and tearDownModule

These should be implemented as functions:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

If an exception is raised in a setUpModule then none of the tests in the module will be run and the tearDownModule will not be run. If the exception is a SkipTest exception then the module will be reported as having been skipped instead of as an error.

25.3.9 Signal Handling

The -c/--catch command-line option to unittest, along with the catchbreak parameter to unittest.main(), provide more friendly handling of control-C during a test run. With catch break behavior enabled control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a KeyboardInterrupt in the usual way.

The control-c handling signal handler attempts to remain compatible with code or tests that install their own signal.SIGINT handler. If the unittest handler is called but isn't the installed signal.SIGINT handler, i.e. it has been replaced by the system under test and delegated to, then it calls the default handler. This will normally be the expected behavior by code that replaces an installed handler and delegates to it. For individual tests that need unittest control-c handling disabled the removeHandler() decorator can be used.

There are a few utility functions for framework authors to enable control-c handling functionality within test frameworks.

unittest.installHandler()
: Install the control-c handler. When a signal.SIGINT is received (usually in response to the user pressing control-c) all registered results have stop() called.

  New in version 2.7.

unittest.registerResult(result)
: Register a TestResult object for control-c handling. Registering a result stores a weak reference to it, so it doesn't prevent the result from being garbage collected.

  Registering a TestResult object has no side-effects if control-c handling is not enabled, so test frameworks can unconditionally register all results they create independently of whether or not handling is enabled.

  New in version 2.7.

unittest.removeResult(result)
: Remove a registered result. Once a result has been removed then stop() will no longer be called on that result object in response to a control-c.

  New in version 2.7.

unittest.removeHandler(function=None)
: When called without arguments this function removes the control-c handler if it has been installed. This function can also be used as a test decorator to temporarily remove the handler while the test is being executed:

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

  New in version 2.7.

25.4 2to3 - Automated Python 2 to 3 code translation

2to3 is a Python program that reads Python 2.x source code and applies a series of fixers to transform it into valid Python 3.x code. The standard library contains a rich set of fixers that will handle almost all code. 2to3 supporting library lib2to3 is, however, a flexible and generic library, so it is possible to write your own fixers for 2to3. lib2to3 could also be adapted to custom applications in which Python code needs to be edited automatically.

25.4.1 Using 2to3

2to3 will usually be installed with the Python interpreter as a script. It is also located in the Tools/scripts directory of the Python root.

2to3's basic arguments are a list of files or directories to transform. The directories are recursively traversed for Python sources.

Here is a sample Python 2.x source file, example.py:

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

It can be converted to Python 3.x code via 2to3 on the command line:

```
$ 2to3 example.py
```

A diff against the original source file is printed. 2to3 can also write the needed modifications right back to the source file. (A backup of the original file is made unless -n is also given.) Writing the changes back is enabled with the -w flag:

```
$ 2to3 -w example.py
```

After transformation, example.py looks like this:

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

Comments and exact indentation are preserved throughout the translation process.

By default, 2to3 runs a set of predefined fixers. The -l flag lists all available fixers. An explicit set of fixers to run can be given with -f. Likewise the -x explicitly disables a fixer. The following example runs only the imports and has_key fixers:

```
$ 2to3 -f imports -f has_key example.py
```

This command runs every fixer except the apply fixer:

```
$ 2to3 -x apply example.py
```

Some fixers are explicit, meaning they aren't run by default and must be listed on the command line to be run. Here, in addition to the default fixers, the idioms fixer is run:

```
$ 2to3 -f all -f idioms example.py
```

Notice how passing all enables all default fixers.

Sometimes 2to3 will find a place in your source code that needs to be changed, but 2to3 cannot fix automatically. In this case, 2to3 will print a warning beneath the diff for a file. You should address the warning in order to have compliant 3.x code.

2to3 can also refactor doctests. To enable this mode, use the -d flag. Note that only doctests will be refactored. This also doesn't require the module to be valid Python. For example, doctest like examples in a reST document could also be refactored with this option.

The -v option enables output of more information on the translation process.

Since some print statements can be parsed as function calls or statements, 2to3 cannot always read files containing the print function. When 2to3 detects the presence of the from __future__ import print_function compiler directive, it modifies its internal grammar to interpret print() as a function. This change can also

be enabled manually with the -p flag. Use -p to run fixers on code that already has had its print statements converted.

The -o or --output-dir option allows specification of an alternate directory for processed output files to be written to. The -n flag is required when using this as backup files do not make sense when not overwriting the input files.

New in version 2.7.3: The -o option was added.

The -W or --write-unchanged-files flag tells 2to3 to always write output files even if no changes were required to the file. This is most useful with -o so that an entire Python source tree is copied with translation from one directory to another. This option implies the -w flag as it would not make sense otherwise.

New in version 2.7.3: The -W flag was added.

The --add-suffix option specifies a string to append to all output filenames. The -n flag is required when specifying this as backups are not necessary when writing to different filenames. Example:

```
$ 2to3 -n -W --add-suffix=3 example.py
```

Will cause a converted file named example.py3 to be written.

New in version 2.7.3: The --add-suffix option was added.

To translate an entire project from one directory tree to another use:

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```

25.4.2 Fixers

Each step of transforming code is encapsulated in a fixer. The command 2to3 -l lists them. As documented above, each can be turned on and off individually. They are described here in more detail.

apply
: Removes usage of apply(). For example apply(function, *args, **kwargs) is converted to function(*args, **kwargs).

asserts
: Replaces deprecated unittest method names with the correct ones.

From	To
failUnlessEqual(a, b)	assertEqual(a, b)
assertEquals(a, b)	assertEqual(a, b)
failIfEqual(a, b)	assertNotEqual(a, b)
assertNotEquals(a, b)	assertNotEqual(a, b)
failUnless(a)	assertTrue(a)
assert_(a)	assertTrue(a)
failIf(a)	assertFalse(a)
failUnlessRaises(exc, cal)	assertRaises(exc, cal)
failUnlessAlmostEqual(a, b)	assertAlmostEqual(a, b)
assertAlmostEquals(a, b)	assertAlmostEqual(a, b)
failIfAlmostEqual(a, b)	assertNotAlmostEqual(a, b)
assertNotAlmostEquals(a, b)	assertNotAlmostEqual(a, b)

basestring
: Converts basestring to str.

buffer
: Converts buffer to memoryview. This fixer is optional because the memoryview API is similar but not exactly the same as that of buffer.

dict
: Fixes dictionary iteration methods. dict.iteritems() is converted to dict.items(), dict.iterkeys() to dict.keys(), and dict.itervalues() to dict.values(). Similarly, dict.viewitems(), dict.viewkeys() and dict.viewvalues() are converted respectively to dict.items(), dict.keys() and dict.values(). It also wraps existing usages of dict.items(), dict.keys(), and dict.values() in a call to list.

except
: Converts except X, T to except X as T.

exec
: Converts the exec statement to the exec() function.

execfile
: Removes usage of execfile(). The argument to execfile() is wrapped in calls to open(), compile(), and exec().

exitfunc
: Changes assignment of sys.exitfunc to use of the atexit module.

filter
: Wraps filter() usage in a list call.

funcattrs
: Fixes function attributes that have been renamed. For example, my_function.func_closure is converted to my_function.__closure__.

future
: Removes from __future__ import new_feature statements.

getcwdu
: Renames os.getcwdu() to os.getcwd().

has_key
: Changes dict.has_key(key) to key in dict.

idioms
: This optional fixer performs several transformations that make Python code more idiomatic. Type comparisons like type(x) is SomeClass and type(x) == SomeClass are converted to isinstance(x, SomeClass). while 1 becomes while True. This fixer also tries to make use of sorted() in appropriate places. For example, this block

```
L = list(some_iterable)
L.sort()
```

  is changed to

```
L = sorted(some_iterable)
```

import
: Detects sibling imports and converts them to relative imports.

imports
: Handles module renames in the standard library.

imports2
: Handles other modules renames in the standard library. It is separate from the imports fixer only because of technical limitations.

input
: Converts input(prompt) to eval(input(prompt)).

intern
: Converts intern() to sys.intern().

isinstance
: Fixes duplicate types in the second argument of isinstance(). For example, isinstance(x, (int, int)) is converted to isinstance(x, (int)).

itertools_imports
: Removes imports of itertools.ifilter(), itertools.izip(), and itertools.imap(). Imports of itertools.ifilterfalse() are also changed to itertools.filterfalse().

itertools
: Changes usage of itertools.ifilter(), itertools.izip(), and itertools.imap() to their built-in equivalents. itertools.ifilterfalse() is changed to itertools.filterfalse().

long
: Renames long to int.

map
: Wraps map() in a list call. It also changes map(None, x) to list(x). Using from future_builtins import map disables this fixer.

metaclass
: Converts the old metaclass syntax (__metaclass__ = Meta in the class body) to the new (class X(metaclass=Meta)).

methodattrs
: Fixes old method attribute names. For example, meth.im_func is converted to meth.__func__.

ne
: Converts the old not-equal syntax, <>, to !=.

next
: Converts the use of iterator's next() methods to the next() function. It also renames next() methods to __next__().

nonzero
: Renames __nonzero__() to __bool__().

numliterals
: Converts octal literals into the new syntax.

paren
: Add extra parenthesis where they are required in list comprehensions. For example, [x for x in 1, 2] becomes [x for x in (1, 2)].

print
: Converts the print statement to the print() function.

raise
: Converts raise E, V to raise E(V), and raise E, V, T to raise E(V).with_traceback(T). If E is a tuple, the translation will be incorrect because substituting tuples for exceptions has been removed in Python 3.

raw_input
: Converts raw_input() to input().

reduce
: Handles the move of reduce() to functools.reduce().

renames
: Changes sys.maxint to sys.maxsize.

repr
: Replaces backtick repr with the repr() function.

set_literal
: Replaces use of the set constructor with set literals. This fixer is optional.

standarderror
: Renames StandardError to Exception.

sys_exc
: Changes the deprecated sys.exc_value, sys.exc_type, sys.exc_traceback to use sys.exc_info().

throw
: Fixes the API change in generator's throw() method.

tuple_params
: Removes implicit tuple parameter unpacking. This fixer inserts temporary variables.

types
: Fixes code broken from the removal of some members in the types module.

unicode
: Renames unicode to str.

urllib
: Handles the rename of urllib and urllib2 to the urllib package.

ws_comma
: Removes excess whitespace from comma separated items. This fixer is optional.

xrange
: Renames xrange() to range() and wraps existing range() calls with list.

xreadlines
: Changes for x in file.xreadlines() to for x in file.

zip
: Wraps zip() usage in a list call. This is disabled when from future_builtins import zip appears.

25.4.3 lib2to3 - 2to3's library

Note: The lib2to3 API should be considered unstable and may change drastically in the future.

25.5 test — Regression tests package for Python

Note: The test package is meant for internal use by Python only. It is documented for the benefit of the core developers of Python. Any use of this package outside of Python's standard library is discouraged as code mentioned here can change or be removed without notice between releases of Python.

The test package contains all regression tests for Python as well as the modules test.support and test.regrtest. test.support is used to enhance your tests while test.regrtest drives the testing suite.

Each module in the test package whose name starts with test_ is a testing suite for a specific module or feature. All new tests should be written using the unittest or doctest module. Some older tests are written using a "traditional" testing style that compares output printed to sys.stdout; this style of test is considered deprecated.

See also:

Module unittest Writing PyUnit regression tests.

Module doctest Tests embedded in documentation strings.

25.5.1 Writing Unit Tests for the test package

It is preferred that tests that use the unittest module follow a few guidelines. One is to name the test module by starting it with test_ and end it with the name of the module being tested. The test methods in the test module should start with test_ and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as # Tests function returns only True or False) should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

def test_main():
    support.run_unittest(MyTestCase1,
                         MyTestCase2,
                         ... list other tests ...
                         )
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    test_main()
```

This boilerplate code allows the testing suite to be run by test.regrtest as well as on its own as a script.

The goal for regression testing is to try to break code. This leads to a few guidelines to be followed:

- The testing suite should exercise all classes, functions, and constants. This includes not just the external API that is to be presented to the outside world but also "private" code.
- Whitebox testing (examining the code being tested when the tests are being written) is preferred. Blackbox testing (testing only the published user interface) is not complete enough to make sure all boundary and edge cases are tested.
- Make sure all possible values are tested including invalid ones. This makes sure that not only all valid values are acceptable but also that improper values are handled correctly.
- Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.
- Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.
- Make sure to clean up after your tests (such as close and remove all temporary files).
- If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.
- Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.
- Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input:

```
class TestFuncAcceptsSequences(unittest.TestCase):

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequences):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequences):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequences):
    arg = (1, 2, 3)
```

See also:

Test Driven Development A book by Kent Beck on writing tests before code.

25.5.2 Running tests using the command-line interface

The test.regrtest module can be run as a script to drive Python's regression test suite, thanks to the -m option: python -m test.regrtest. Running the script by itself automatically starts running all regression tests in the test package. It does this by finding all modules in the package whose name starts with test_,

importing them, and executing the function test_main() if present. The names of tests to execute may also be passed to the script. Specifying a single regression test (python -m test.regrtest test_spam) will minimize output and only print whether the test passed or failed and thus minimize output.

Running test.regrtest directly allows what resources are available for tests to use to be set. You do this by using the -u command-line option. Specifying all as the value for the -u option enables all possible resources: python -m test.regrtest -uall. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after all. The command python -m test.regrtest -uall,-audio,-largefile will run test.regrtest with all resources except the audio and largefile resources. For a list of all resources and more command-line options, run python -m test.regrtest -h.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On Unix, you can run make test at the top-level directory where Python was built. On Windows, executing rt.bat from your PCBuild directory will run all regression tests.

Changed in version 2.7.14: The test package can be run as a script: python -m test. This works the same as running the test.regrtest module.

25.6 test.support — Utility functions for tests

Note: The test.test_support module has been renamed to test.support in Python 3.x and 2.7.14. The name test.test_support has been retained as an alias in 2.7.

The test.support module provides support for Python's regression tests.

This module defines the following exceptions:

exception test.support.TestFailed
: Exception to be raised when a test fails. This is deprecated in favor of unittest-based tests and unittest.TestCase's assertion methods.

exception test.support.ResourceDenied
: Subclass of unittest.SkipTest. Raised when a resource (such as a network connection) is not available. Raised by the requires() function.

The test.support module defines the following constants:

test.support.verbose
: True when verbose output is enabled. Should be checked when more detailed information is desired about a running test. verbose is set by test.regrtest.

test.support.have_unicode
: True when Unicode support is available.

test.support.is_jython
: True if the running interpreter is Jython.

test.support.TESTFN
: Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

The test.support module defines the following functions:

test.support.forget(module_name)
: Remove the module named module_name from sys.modules and delete any byte-compiled files of the module.

test.support.is_resource_enabled(resource)
: Return True if resource is enabled and available. The list of available resources is only set when test.regrtest is executing the tests.

test.support.requires(resource[, msg])
: Raise ResourceDenied if resource is not available. msg is the argument to ResourceDenied if it is raised. Always returns True if called by a function whose __name__ is '__main__'. Used when tests are executed by test.regrtest.

test.support.findfile(filename)
: Return the path to the file named filename. If no match is found filename is returned. This does not equal a failure since it could be the path to the file.

test.support.run_unittest(*classes)
: Execute unittest.TestCase subclasses passed to the function. The function scans the classes for methods starting with the prefix test_ and executes the tests individually.

  It is also legal to pass strings as parameters; these should be keys in sys.modules. Each associated module will be scanned by unittest.TestLoader.loadTestsFromModule(). This is usually seen in the following test_main() function:

```
def test_main():
    support.run_unittest(__name__)
```

  This will run all tests defined in the named module.

test.support.check_warnings(*filters, quiet=True)
: A convenience wrapper for warnings.catch_warnings() that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling warnings.catch_warnings(record=True) with warnings.simplefilter() set to always and with the option to automatically validate the results that are recorded.

  check_warnings accepts 2-tuples of the form ("message regexp", WarningCategory) as positional arguments. If one or more filters are provided, or if the optional keyword argument quiet is False, it checks to make sure the warnings are as expected: each specified filter must match at least one of the warnings raised by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set quiet to True.

  If no arguments are specified, it defaults to:

```
check_warnings(("", Warning), quiet=True)
```

  In this case all warnings are caught and no errors are raised.

  On entry to the context manager, a WarningRecorder instance is returned. The underlying warnings list from catch_warnings() is available via the recorder object's warnings attribute. As a convenience, the attributes of the object representing the most recent warning can also be accessed directly through the recorder object (see example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return None.

  The recorder object also has a reset() method, which clears the warnings list.

  The context manager is designed to be used like this:

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

In this case if either warning was not raised, or some other warning was raised, check_warnings() would raise an error.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used:

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

Here all warnings will be caught, and the test code tests the captured warnings directly.

New in version 2.6.

Changed in version 2.7: New optional arguments filters and quiet.

test.support.check_py3k_warnings(*filters, quiet=False)

Similar to check_warnings(), but for Python 3 compatibility warnings. If sys.py3kwarning == 1, it checks if the warning is effectively raised. If sys.py3kwarning == 0, it checks that no warning is raised. It accepts 2-tuples of the form ("message regexp", WarningCategory) as positional arguments. When the optional keyword argument quiet is True, it does not fail if a filter catches nothing. Without arguments, it defaults to:

```
check_py3k_warnings(("", DeprecationWarning), quiet=False)
```

New in version 2.7.

test.support.captured_stdout()

This is a context manager that runs the with statement body using a StringIO.StringIO object as sys.stdout. That object can be retrieved using the as clause of the with statement.

Example use:

```
with captured_stdout() as s:
    print "hello"
assert s.getvalue() == "hello\n"
```

New in version 2.6.

test.support.import_module(name, deprecated=False)

This function imports and returns the named module. Unlike a normal import, this function raises unittest.SkipTest if the module cannot be imported.

Module and package deprecation messages are suppressed during this import if deprecated is True.

New in version 2.7.

test.support.import_fresh_module(name, fresh=(), blocked=(), deprecated=False)

This function imports and returns a fresh copy of the named Python module by removing the named module from sys.modules before doing the import. Note that unlike reload(), the original module is not affected by this operation.

fresh is an iterable of additional module names that are also removed from the sys.modules cache before doing the import.

blocked is an iterable of module names that are replaced with 0 in the module cache during the import to ensure that attempts to import them raise ImportError.

The named module and any modules named in the fresh and blocked parameters are saved before starting the import and then reinserted into sys.modules when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if deprecated is True.

This function will raise unittest.SkipTest if the named module cannot be imported.

Example use:

```
# Get copies of the warnings module for testing without
# affecting the version being used by the rest of the test suite
# One copy uses the C implementation, the other is forced to use
# the pure Python fallback implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

New in version 2.7.

The test.support module defines the following classes:

class test.support.TransientResource(exc[, **kwargs])
Instances are a context manager that raises ResourceDenied if the specified exception type is raised. Any keyword arguments are treated as attribute/value pairs to be compared against any exception raised within the with statement. Only if all pairs match properly against attributes on the exception is ResourceDenied raised.

New in version 2.6.

class test.support.EnvironmentVarGuard
Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete dictionary interface for querying/modifying the underlying os.environ. After exit from the context manager all changes to environment variables done through this instance will be rolled back.

New in version 2.6.

Changed in version 2.7: Added dictionary interface.

EnvironmentVarGuard.set(envvar, value)
Temporarily set the environment variable envvar to the value of value.

EnvironmentVarGuard.unset(envvar)
Temporarily unset the environment variable envvar.

class test.support.WarningsRecorder
Class used to record warnings for unit tests. See documentation of check_warnings() above for more details.

New in version 2.6.

CHAPTER

TWENTYSIX

DEBUGGING AND PROFILING

These libraries help you with Python development: the debugger enables you to step through code, analyze stack frames and set breakpoints etc., and the profilers run code and give you a detailed breakdown of execution times, allowing you to identify bottlenecks in your programs.

26.1 bdb — Debugger framework

Source code: Lib/bdb.py

The bdb module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

The following exception is defined:

exception bdb.BdbQuit
: Exception raised by the Bdb class for quitting the debugger.

The bdb module also defines two classes:

class bdb.Breakpoint(self, file, line, temporary=0, cond=None, funcname=None)
: This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

  Breakpoints are indexed by number through a list called bpbynumber and by (file, line) pairs through bplist. The former points to a single instance of class Breakpoint. The latter points to a list of such instances since there may be more than one breakpoint per line.

  When creating a breakpoint, its associated filename should be in canonical form. If a funcname is defined, a breakpoint hit will be counted when the first line of that function is executed. A conditional breakpoint always counts a hit.

  Breakpoint instances have the following methods:

  deleteMe()
  : Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

  enable()
  : Mark the breakpoint as enabled.

  disable()
  : Mark the breakpoint as disabled.

  pprint([out])
  : Print all the information about the breakpoint:

- The breakpoint number.
- If it is temporary or not.
- Its file,line position.
- The condition that causes a break.
- If it must be ignored the next N times.
- The breakpoint hit count.

class bdb.Bdb(skip=None)

The Bdb class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (pdb.Pdb) is an example.

The skip argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the __name__ in the frame globals.

New in version 2.7: The skip argument.

The following methods of Bdb normally don't need to be overridden.

canonic(filename)

Auxiliary method for getting a filename in a canonical form, that is, as a case-normalized (on case-insensitive filesystems) absolute path, stripped of surrounding angle brackets.

reset()

Set the botframe, stopframe, returnframe and quitting attributes with values ready to start debugging.

trace_dispatch(frame, event, arg)

This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. event can be one of the following:

- "line": A new line of code is going to be executed.
- "call": A function is about to be called, or another code block entered.
- "return": A function or other code block is about to return.
- "exception": An exception has occurred.
- "c_call": A C function is about to be called.
- "c_return": A C function has returned.
- "c_exception": A C function has raised an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

The arg parameter depends on the previous event.

See the documentation for sys.settrace() for more information on the trace function. For more information on code and frame objects, refer to types.

dispatch_line(frame)
: If the debugger should stop on the current line, invoke the user_line() method (which should be overridden in subclasses). Raise a BdbQuit exception if the Bdb.quitting flag is set (which can be set from user_line()). Return a reference to the trace_dispatch() method for further tracing in that scope.

dispatch_call(frame, arg)
: If the debugger should stop on this function call, invoke the user_call() method (which should be overridden in subclasses). Raise a BdbQuit exception if the Bdb.quitting flag is set (which can be set from user_call()). Return a reference to the trace_dispatch() method for further tracing in that scope.

dispatch_return(frame, arg)
: If the debugger should stop on this function return, invoke the user_return() method (which should be overridden in subclasses). Raise a BdbQuit exception if the Bdb.quitting flag is set (which can be set from user_return()). Return a reference to the trace_dispatch() method for further tracing in that scope.

dispatch_exception(frame, arg)
: If the debugger should stop at this exception, invokes the user_exception() method (which should be overridden in subclasses). Raise a BdbQuit exception if the Bdb.quitting flag is set (which can be set from user_exception()). Return a reference to the trace_dispatch() method for further tracing in that scope.

Normally derived classes don't override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

stop_here(frame)
: This method checks if the frame is somewhere below botframe in the call stack. botframe is the frame in which debugging started.

break_here(frame)
: This method checks if there is a breakpoint in the filename and line belonging to frame or, at least, in the current function. If the breakpoint is a temporary one, this method deletes it.

break_anywhere(frame)
: This method checks if there is a breakpoint in the filename of the current frame.

Derived classes should override these methods to gain control over debugger operation.

user_call(frame, argument_list)
: This method is called from dispatch_call() when there is the possibility that a break might be necessary anywhere inside the called function.

user_line(frame)
: This method is called from dispatch_line() when either stop_here() or break_here() yields True.

user_return(frame, return_value)
: This method is called from dispatch_return() when stop_here() yields True.

user_exception(frame, exc_info)
: This method is called from dispatch_exception() when stop_here() yields True.

do_clear(arg)
: Handle how a breakpoint must be removed when it is a temporary one.

  This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

set_step()
: Stop after one line of code.

set_next(frame)
Stop on the next line in or below the given frame.

set_return(frame)
Stop when returning from the given frame.

set_until(frame)
Stop when the line with the line no greater than the current one is reached or when returning from current frame.

set_trace([frame])
Start debugging from frame. If frame is not specified, debugging starts from caller's frame.

set_continue()
Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to None.

set_quit()
Set the quitting attribute to True. This raises BdbQuit in the next call to one of the dispatch_*() methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or None if all is well.

set_break(filename, lineno, temporary=0, cond=None, funcname=None)
Set a new breakpoint. If the lineno line doesn't exist for the filename passed as argument, return an error message. The filename should be in canonical form, as described in the canonic() method.

clear_break(filename, lineno)
Delete the breakpoints in filename and lineno. If none were set, an error message is returned.

clear_bpbynumber(arg)
Delete the breakpoint which has the index arg in the Breakpoint.bpbynumber. If arg is not numeric or out of range, return an error message.

clear_all_file_breaks(filename)
Delete all breakpoints in filename. If none were set, an error message is returned.

clear_all_breaks()
Delete all existing breakpoints.

get_break(filename, lineno)
Check if there is a breakpoint for lineno of filename.

get_breaks(filename, lineno)
Return all breakpoints for lineno in filename, or an empty list if none are set.

get_file_breaks(filename)
Return all breakpoints in filename, or an empty list if none are set.

get_all_breaks()
Return all breakpoints that are set.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

get_stack(f, t)
Get a list of records for a frame and all higher (calling) and lower frames, and the size of the higher part.

format_stack_entry(frame_lineno[, lprefix=': '])
Return a string with information about a stack entry, identified by a (frame, lineno) tuple:

- The canonical form of the filename which contains the frame.
- The function name, or "<lambda>".
- The input arguments.
- The return value.
- The line of code (if it exists).

The following two methods can be called by clients to use a debugger to debug a statement, given as a string.

run(cmd[, globals[, locals]])
: Debug a statement executed via the exec statement. globals defaults to __main__.__dict__, locals defaults to globals.

runeval(expr[, globals[, locals]])
: Debug an expression executed via the eval() function. globals and locals have the same meaning as in run().

runctx(cmd, globals, locals)
: For backwards compatibility. Calls the run() method.

runcall(func, *args, **kwds)
: Debug a single function call, and return its result.

Finally, the module defines the following functions:

bdb.checkfuncname(b, frame)
: Check whether we should break here, depending on the way the breakpoint b was set.

  If it was set via line number, it checks if b.line is the same as the one in the frame also passed as argument. If the breakpoint was set via function name, we have to check we are in the right frame (the right function) and if we are in its first executable line.

bdb.effective(file, line, frame)
: Determine if there is an effective (active) breakpoint at this line of code. Return a tuple of the breakpoint and a boolean that indicates if it is ok to delete a temporary breakpoint. Return (None, None) if there is no matching breakpoint.

bdb.set_trace()
: Start debugging with a Bdb instance from caller's frame.

26.2 pdb — The Python Debugger

Source code: Lib/pdb.py

The module pdb defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger is extensible — it is actually defined as the class Pdb. This is currently undocumented but easily understood by reading the source. The extension interface uses the modules bdb and cmd.

The debugger's prompt is (Pdb). Typical usage to run a program under control of the debugger is:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

pdb.py can also be invoked as a script to debug other scripts. For example:

```
python -m pdb myscript.py
```

When invoked as a script, pdb will automatically enter post-mortem debugging if the program being debugged exits abnormally. After post-mortem debugging (or after normal exit of the program), pdb will restart the program. Automatic restarting preserves pdb's state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program's exit.

New in version 2.4: Restarting post-mortem behavior added.

The typical usage to break into the debugger from a running program is to insert

```
import pdb; pdb.set_trace()
```

at the location you want to break into the debugger. You can then step through the code following this statement, and continue running without the debugger using the c command.

The typical usage to inspect a crashed program is:

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print spam
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print spam
(Pdb)
```

The module defines the following functions; each enters the debugger in a slightly different way:

pdb.run(statement[, globals[, locals]])
: Execute the statement (given as a string) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type continue, or you can step through the statement using step or next (all these commands are explained below). The optional globals and locals arguments specify the environment in which the code is executed; by default the dictionary of the module __main__ is used. (See the explanation of the exec statement or the eval() built-in function.)

pdb.runeval(expression[, globals[, locals]])
: Evaluate the expression (given as a string) under debugger control. When runeval() returns, it returns

the value of the expression. Otherwise this function is similar to run().

pdb.runcall(function[, argument, ...])
Call the function (a function or method object, not a string) with the given arguments. When runcall() returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

pdb.set_trace()
Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).

pdb.post_mortem([traceback])
Enter post-mortem debugging of the given traceback object. If no traceback is given, it uses the one of the exception that is currently being handled (an exception must be being handled if the default is to be used).

pdb.pm()
Enter post-mortem debugging of the traceback found in sys.last_traceback.

The run* functions and set_trace() are aliases for instantiating the Pdb class and calling the method of the same name. If you want to access further features, you have to do this yourself:

class pdb.Pdb(completekey='tab', stdin=None, stdout=None, skip=None)
Pdb is the debugger class.

The completekey, stdin and stdout arguments are passed to the underlying cmd.Cmd class; see the description there.

The skip argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns.[1]

Example call to enable tracing with skip:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

New in version 2.7: The skip argument.

run(statement[, globals[, locals]])
runeval(expression[, globals[, locals]])
runcall(function[, argument, ...])
set_trace()
See the documentation for the functions explained above.

26.3 Debugger Commands

The debugger recognizes the following commands. Most commands can be abbreviated to one or two letters; e.g. h(elp) means that either h or help can be used to enter the help command (but not he or hel, nor H or Help or HELP). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets ([]) in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar (|).

Entering a blank line repeats the last command entered. Exception: if the last command was a list command, the next 11 lines are listed.

Commands that the debugger doesn't recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation

[1] Whether a frame is considered to originate in a certain module is determined by the __name__ in the frame globals.

point (!). This is a powerful way to inspect the program being debugged; it is even possible to change a variable or call a function. When an exception occurs in such a statement, the exception name is printed but the debugger's state is not changed.

Multiple commands may be entered on a single line, separated by ;;. (A single ; is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first ;; pair, even if it is in the middle of a quoted string.

The debugger supports aliases. Aliases can have parameters which allows one a certain level of adaptability to the context under examination.

If a file .pdbrc exists in the user's home directory or in the current directory, it is read in and executed as if it had been typed at the debugger prompt. This is particularly useful for aliases. If both files exist, the one in the home directory is read first and aliases defined there can be overridden by the local file.

h(elp) [command] Without argument, print the list of available commands. With a command as argument, print help about that command. help pdb displays the full documentation file; if the environment variable PAGER is defined, the file is piped through that command instead. Since the command argument must be an identifier, help exec must be entered to get help on the ! command.

w(here) Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.

d(own) Move the current frame one level down in the stack trace (to a newer frame).

u(p) Move the current frame one level up in the stack trace (to an older frame).

b(reak) [[filename:]lineno | function[, condition]] With a lineno argument, set a break there in the current file. With a function argument, set a break at the first executable statement within that function. The line number may be prefixed with a filename and a colon, to specify a breakpoint in another file (probably one that hasn't been loaded yet). The file is searched on sys.path. Note that each breakpoint is assigned a number to which all the other breakpoint commands refer.

If a second argument is present, it is an expression which must evaluate to true before the breakpoint is honored.

Without argument, list all breaks, including for each breakpoint, the number of times that breakpoint has been hit, the current ignore count, and the associated condition if any.

tbreak [[filename:]lineno | function[, condition]] Temporary breakpoint, which is removed automatically when it is first hit. The arguments are the same as break.

cl(ear) [filename:lineno | bpnumber [bpnumber ...]] With a filename:lineno argument, clear all the breakpoints at this line. With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation).

disable [bpnumber [bpnumber ...]] Disables the breakpoints given as a space separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re-)enabled.

enable [bpnumber [bpnumber ...]] Enables the breakpoints specified.

ignore bpnumber [count] Sets the ignore count for the given breakpoint number. If count is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the count is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

condition bpnumber [condition] Condition is an expression which must evaluate to true before the breakpoint is honored. If condition is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.

commands [bpnumber] Specify a list of commands for breakpoint number bpnumber. The commands themselves appear on the following lines. Type a line containing just 'end' to terminate the commands. An example:

```
(Pdb) commands 1
(com) print some_variable
(com) end
(Pdb)
```

To remove all commands from a breakpoint, type commands and follow it immediately with end; that is, give no commands.

With no bpnumber argument, commands refers to the last breakpoint set.

You can use breakpoint commands to start your program up again. Simply use the continue command, or step, or any other command that resumes execution.

Specifying any command resuming execution (currently continue, step, next, return, jump, quit and their abbreviations) terminates the command list (as if that command was immediately followed by end). This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

If you use the 'silent' command in the command list, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the other commands print anything, you see no sign that the breakpoint was reached.

New in version 2.5.

s(tep) Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

n(ext) Continue execution until the next line in the current function is reached or it returns. (The difference between next and step is that step stops inside a called function, while next executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

unt(il) Continue execution until the line with the line number greater than the current one is reached or when returning from current frame.

New in version 2.6.

r(eturn) Continue execution until the current function returns.

c(ont(inue)) Continue execution, only stop when a breakpoint is encountered.

j(ump) lineno Set the next line that will be executed. Only available in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don't want to run.

It should be noted that not all jumps are allowed — for instance it is not possible to jump into the middle of a for loop or out of a finally clause.

l(ist) [first[, last]] List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With one argument, list 11 lines around at that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

a(rgs) Print the argument list of the current function.

p expression Evaluate the expression in the current context and print its value.

Note: print can also be used, but is not a debugger command — this executes the Python print statement.

pp expression Like the p command, except the value of the expression is pretty-printed using the pprint module.

alias [name [command]] Creates an alias called name that executes command. The command must not be enclosed in quotes. Replaceable parameters can be indicated by %1, %2, and so on, while %* is replaced by all the parameters. If no command is given, the current alias for name is shown. If no arguments are given, all aliases are listed.

Aliases may be nested and can contain anything that can be legally typed at the pdb prompt. Note that internal pdb commands can be overridden by aliases. Such a command is then hidden until the alias is removed. Aliasing is recursively applied to the first word of the command line; all other words in the line are left alone.

As an example, here are two useful aliases (especially when placed in the .pdbrc file):

```
#Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.__dict__[k]
#Print instance variables in self
alias ps pi self
```

unalias name Deletes the specified alias.

[!]statement Execute the (one-line) statement in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command. To set a global variable, you can prefix the assignment command with a global command on the same line, e.g.:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...] Restart the debugged Python program. If an argument is supplied, it is split with "shlex" and the result is used as the new sys.argv. History, breakpoints, actions and debugger options are preserved. "restart" is an alias for "run".

New in version 2.6.

q(uit) Quit from the debugger. The program being executed is aborted.

26.4 The Python Profilers

Source code: Lib/profile.py and Lib/pstats.py

26.4.1 Introduction to the profilers

cProfile and profile provide deterministic profiling of Python programs. A profile is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the pstats module.

The Python standard library provides three different implementations of the same profiling interface:

1. cProfile is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on lsprof, contributed by Brett Rosen and Ted Czotter.

   New in version 2.5.

2. profile, a pure Python module whose interface is imitated by cProfile, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

   Changed in version 2.4: Now also reports the time spent in calls to built-in functions and methods.

3. hotshot was an experimental C module that focused on minimizing the overhead of profiling, at the expense of longer data post-processing times. It is no longer maintained and may be dropped in a future version of Python.

   Changed in version 2.5: The results should be more meaningful than in the past: the timing core contained a critical bug.

The profile and cProfile modules export the same interface, so they are mostly interchangeable; cProfile has a much lower overhead but is newer and might not be available on all systems. cProfile is really a compatibility layer on top of the internal _lsprof module. The hotshot module is reserved for specialized usage.

Note: The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes (for that, there is timeit for reasonably accurate results). This particularly applies to benchmarking Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python one.

26.4.2 Instant User's Manual

This section is provided for users that "don't want to read the manual." It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile a function that takes a single argument, you can do:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(Use profile instead of cProfile if the latter is not available on your system.)

The above action would run re.compile() and print profile results like the following:

```
      197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

The first line indicates that 197 calls were monitored. Of those calls, 192 were primitive, meaning that the call was not induced via recursion. The next line: Ordered by: standard name, indicates that the text string in the far right column was used to sort the output. The column headings include:

ncalls for the number of calls,

tottime for the total time spent in the given function (and excluding time made in calls to sub-functions)

percall is the quotient of tottime divided by ncalls

cumtime is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.

percall is the quotient of cumtime divided by primitive calls

filename:lineno(function) provides the respective data of each function

When there are two numbers in the first column (for example 3/1), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the run() function:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

The pstats.Stats class reads profile results from a file and formats them in various ways.

The file cProfile can also be invoked as a script to profile another script. For example:

```
python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

-o writes the profile results to a file instead of to stdout

-s specifies one of the sort_stats() sort values to sort the output by. This only applies when -o is not supplied.

The pstats module's Stats class has a variety of methods for manipulating and printing the data saved into a profile results file:

```
import pstats
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The strip_dirs() method removed the extraneous path from all the module names. The sort_stats() method sorted all the entries according to the standard module/line/name string that is printed. The print_stats() method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats('cumulative').print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats('time').print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```
p.sort_stats('file').print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with __init__ in them). As one final example, you could try:

```
p.sort_stats('time', 'cum').print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: .5) of its original size, then only lines containing init are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (p is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do:

```
p.print_callees()
p.add('restats')
```

Invoked as a script, the pstats module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using cmd) and interactive help.

26.4.3 profile and cProfile Module Reference

Both the profile and cProfile modules provide the following functions:

profile.run(command, filename=None, sort=-1)
: This function takes a single argument that can be passed to the exec() function, and an optional file name. In all cases this routine executes:

  ```
  exec(command, __main__.__dict__, __main__.__dict__)
  ```

  and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a Stats instance and prints a simple profiling report. If the sort value is specified it is passed to this Stats instance to control how the results are sorted.

profile.runctx(command, globals, locals, filename=None)
: This function is similar to run(), with added arguments to supply the globals and locals dictionaries for the command string. This routine executes:

  ```
  exec(command, globals, locals)
  ```

  and gathers profiling statistics as in the run() function above.

class profile.Profile(timer=None, timeunit=0.0, subcalls=True, builtins=True)
: This class is normally only used if more precise control over profiling is needed than what the cProfile.run() function provides.

A custom timer can be supplied for measuring how long code takes to run via the timer argument. This must be a function that returns a single number representing the current time. If the number is an integer, the timeunit specifies a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be .001.

Directly using the Profile class allows formatting profile results without writing the profile data to a file:

```
import cProfile, pstats, StringIO
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = StringIO.StringIO()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

enable()
: Start collecting profiling data.

disable()
: Stop collecting profiling data.

create_stats()
: Stop collecting profiling data and record the results internally as the current profile.

print_stats(sort=-1)
: Create a Stats object based on the current profile and print the results to stdout.

dump_stats(filename)
: Write the results of the current profile to filename.

run(cmd)
: Profile the cmd via exec().

runctx(cmd, globals, locals)
: Profile the cmd via exec() with the specified global and local environment.

runcall(func, *args, **kwargs)
: Profile func(*args, **kwargs)

26.4.4 The Stats Class

Analysis of the profiler data is done using the Stats class.

class pstats.Stats(*filenames or profile, stream=sys.stdout)
: This class constructor creates an instance of a "statistics object" from a filename (or list of filenames) or from a Profile instance. Output will be printed to the stream specified by stream.

  The file selected by the above constructor must have been created by the corresponding version of profile or cProfile. To be specific, there is no file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing Stats object, the add() method can be used.

Instead of reading the profile data from a file, a cProfile.Profile or profile.Profile object can be used as the profile data source.

Stats objects have the following methods:

strip_dirs()
: This method for the Stats class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If strip_dirs() causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

add(*filenames)
: This method of the Stats class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of profile.run() or cProfile.run(). Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

dump_stats(filename)
: Save the data loaded into the Stats object to a file named filename. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the profile.Profile and cProfile.Profile classes.

New in version 2.3.

sort_stats(*keys)
: This method modifies the Stats object by sorting it according to the supplied criteria. The argument is typically a string identifying the basis of a sort (example: `'time'` or `'name'`).

  When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, sort_stats(`'name'`, `'file'`) will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

  Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

Valid Arg	Meaning
`'calls'`	call count
`'cumulative'`	cumulative time
`'cumtime'`	cumulative time
`'file'`	file name
`'filename'`	file name
`'module'`	file name
`'ncalls'`	call count
`'pcalls'`	primitive call count
`'line'`	line number
`'name'`	function name
`'nfl'`	name/file/line
`'stdname'`	standard name
`'time'`	internal time
`'tottime'`	internal time

  Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle

distinction between 'nfl' and 'stdname' is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, 'nfl' does a numeric compare of the line numbers. In fact, sort_stats('nfl') is the same as sort_stats('name', 'file', 'line').

For backward-compatibility reasons, the numeric arguments -1, 0, 1, and 2 are permitted. They are interpreted as 'stdname', 'calls', 'time', and 'cumulative' respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

reverse_order()
This method for the Stats class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

print_stats(*restrictions)
This method for the Stats class prints out a report as described in the profile.run() definition.

The order of the printing is based on the last sort_stats() operation done on the object (subject to caveats in add() and strip_dirs()).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a regular expression (to pattern match the standard name that is printed. If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename .*foo:. In contrast, the command:

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names .*foo:, and then proceed to only print the first 10% of them.

print_callers(*restrictions)
This method for the Stats class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by print_stats(), and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With profile, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With cProfile, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

print_callees(*restrictions)
This method for the Stats class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the print_callers() method.

26.4.5 What Is Deterministic Profiling?

Deterministic profiling is meant to reflect the fact that all function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, statistical profiling (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a hook (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify "hot loops" that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

26.4.6 Limitations

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying "clock" is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the "error" will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it "takes a while" from when an event is dispatched until the profiler's call to get the time actually gets the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock's value was obtained (and then squirreled away), until the user's code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it can accumulate and become very significant.

The problem is more important with profile than with the lower-overhead cProfile. For this reason, profile provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-).) Do not be alarmed by negative numbers in the profile. They should only appear if you have calibrated your profiler, and the results are actually better than without calibration.

26.4.7 Calibration

The profiler of the profile module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see Limitations).

```
import profile
pr = profile.Profile()
for i in range(5):
    print pr.calibrate(10000)
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running Mac OS X, and using Python's time.clock() as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is very fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it:[1]

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will "less often" show up as negative in profile statistics.

26.4.8 Using a custom timer

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the Profile class constructor:

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call your_time_func. Depending on whether you are using profile.Profile or cProfile.Profile, your_time_func's return value will be interpreted differently:

profile.Profile your_time_func should return a single number, or a list of numbers whose sum is the current time (like what os.times() returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see Calibration). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (os.times() is pretty bad, as it returns a tuple of floating point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

cProfile.Profile your_time_func should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if your_integer_time_func returns times measured in thousands of seconds, you would construct the Profile instance as follows:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

[1] Prior to Python 2.2, it was necessary to edit the profiler source code to embed the bias as a literal number. You still can, but that method is no longer described, because no longer needed.

As the cProfile.Profile class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal _lsprof module.

26.5 hotshot — High performance logging profiler

New in version 2.2.

This module provides a nicer interface to the _hotshot C module. Hotshot is a replacement for the existing profile module. As it's written mostly in C, it should result in a much smaller performance impact than the existing profile module.

Note: The hotshot module focuses on minimizing the overhead while profiling, at the expense of long data post-processing times. For common usage it is recommended to use cProfile instead. hotshot is not maintained and might be removed from the standard library in the future.

Changed in version 2.5: The results should be more meaningful than in the past: the timing core contained a critical bug.

Note: The hotshot profiler does not yet work well with threads. It is useful to use an unthreaded script to run the profiler over the code you're interested in measuring if at all possible.

---

class hotshot.Profile(logfile[, lineevents[, linetimings]])
The profiler object. The argument logfile is the name of a log file to use for logged profile data. The argument lineevents specifies whether to generate events for every source line, or just on function call/return. It defaults to 0 (only log function call/return). The argument linetimings specifies whether to record timing information. It defaults to 1 (store timing information).

26.5.1 Profile Objects

Profile objects have the following methods:

Profile.addinfo(key, value)
Add an arbitrary labelled value to the profile output.

Profile.close()
Close the logfile and terminate the profiler.

Profile.fileno()
Return the file descriptor of the profiler's log file.

Profile.run(cmd)
Profile an exec-compatible string in the script environment. The globals from the __main__ module are used as both the globals and locals for the script.

Profile.runcall(func, *args, **keywords)
Profile a single call of a callable. Additional positional and keyword arguments may be passed along; the result of the call is returned, and exceptions are allowed to propagate cleanly, while ensuring that profiling is disabled on the way out.

Profile.runctx(cmd, globals, locals)
Evaluate an exec-compatible string in a specific environment. The string is compiled before profiling begins.

Profile.start()
Start the profiler.

Profile.stop()
Stop the profiler.

26.5.2 Using hotshot data

New in version 2.2.

This module loads hotshot profiling data into the standard pstats Stats objects.

hotshot.stats.load(filename)
Load hotshot data from filename. Returns an instance of the pstats.Stats class.

See also:

Module profile The profile module's Stats class

26.5.3 Example Usage

Note that this example runs the Python "benchmark" pystones. It can take some time to run, and will produce large output files.

```
>>> import hotshot, hotshot.stats, test.pystone
>>> prof = hotshot.Profile("stones.prof")
>>> benchtime, stones = prof.runcall(test.pystone.pystones)
>>> prof.close()
>>> stats = hotshot.stats.load("stones.prof")
>>> stats.strip_dirs()
>>> stats.sort_stats('time', 'calls')
>>> stats.print_stats(20)
         850004 function calls in 10.090 CPU seconds

   Ordered by: internal time, call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    3.295    3.295   10.090   10.090 pystone.py:79(Proc0)
   150000    1.315    0.000    1.315    0.000 pystone.py:203(Proc7)
    50000    1.313    0.000    1.463    0.000 pystone.py:229(Func2)
.
.
.
```

26.6 timeit — Measure execution time of small code snippets

New in version 2.3.

Source code: Lib/timeit.py

This module provides a simple way to time small bits of Python code. It has both a Command-Line Interface as well as a callable one. It avoids a number of common traps for measuring execution times. See also Tim Peters' introduction to the "Algorithms" chapter in the Python Cookbook, published by O'Reilly.

26.6.1 Basic Examples

The following example shows how the Command-Line Interface can be used to compare three different expressions:

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
$ python -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 33.4 usec per loop
$ python -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 25.2 usec per loop
```

This can be achieved from the Python Interface with:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.7288308143615723
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.5858950614929199
```

Note however that timeit will automatically determine the number of repetitions only when the command-line interface is used. In the Examples section you can find more advanced examples.

26.6.2 Python Interface

The module defines three convenience functions and a public class:

timeit.timeit(stmt='pass', setup='pass', timer=<default timer>, number=1000000)
: Create a Timer instance with the given statement, setup code and timer function and run its timeit() method with number executions.

  New in version 2.6.

timeit.repeat(stmt='pass', setup='pass', timer=<default timer>, repeat=3, number=1000000)
: Create a Timer instance with the given statement, setup code and timer function and run its repeat() method with the given repeat count and number executions.

  New in version 2.6.

timeit.default_timer()
: Define a default timer, in a platform-specific manner. On Windows, time.clock() has microsecond granularity, but time.time()'s granularity is 1/60th of a second. On Unix, time.clock() has 1/100th of a second granularity, and time.time() is much more precise. On either platform, default_timer() measures wall clock time, not the CPU time. This means that other processes running on the same computer may interfere with the timing.

class timeit.Timer(stmt='pass', setup='pass', timer=<timer function>)
: Class for timing execution speed of small code snippets.

  The constructor takes a statement to be timed, an additional statement used for setup, and a timer function. Both statements default to 'pass'; the timer function is platform-dependent (see the module doc string). stmt and setup may also contain multiple statements separated by ; or newlines, as long as they don't contain multi-line string literals.

  To measure the execution time of the first statement, use the timeit() method. The repeat() method is a convenience to call timeit() multiple times and return a list of results.

Changed in version 2.6: The stmt and setup parameters can now also take objects that are callable without arguments. This will embed calls to them in a timer function that will then be executed by timeit(). Note that the timing overhead is a little larger in this case because of the extra function calls.

timeit(number=1000000)
Time number executions of the main statement. This executes the setup statement once, and then returns the time it takes to execute the main statement a number of times, measured in seconds as a float. The argument is the number of times through the loop, defaulting to one million. The main statement, the setup statement and the timer function to be used are passed to the constructor.

Note: By default, timeit() temporarily turns off garbage collection during the timing. The advantage of this approach is that it makes independent timings more comparable. This disadvantage is that GC may be an important component of the performance of the function being measured. If so, GC can be re-enabled as the first statement in the setup string. For example:

```
timeit.Timer('for i in xrange(10): oct(i)', 'gc.enable()').timeit()
```

---

repeat(repeat=3, number=1000000)
Call timeit() a few times.

This is a convenience function that calls the timeit() repeatedly, returning a list of results. The first argument specifies how many times to call timeit(). The second argument specifies the number argument for timeit().

Note: It's tempting to calculate mean and standard deviation from the result vector and report these. However, this is not very useful. In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python's speed, but by other processes interfering with your timing accuracy. So the min() of the result is probably the only number you should be interested in. After that, you should look at the entire vector and apply common sense rather than statistics.

---

print_exc(file=None)
Helper to print a traceback from the timed code.

Typical use:

```
t = Timer(...)       # outside the try/except
try:
    t.timeit(...)    # or t.repeat(...)
except:
    t.print_exc()
```

The advantage over the standard traceback is that source lines in the compiled template will be displayed. The optional file argument directs where the traceback is sent; it defaults to sys.stderr.

26.6.3 Command-Line Interface

When called as a program from the command line, the following form is used:

```
python -m timeit [-n N] [-r N] [-s S] [-t] [-c] [-h] [statement ...]
```

Where the following options are understood:

-n N, --number=N
: how many times to execute 'statement'

-r N, --repeat=N
: how many times to repeat the timer (default 3)

-s S, --setup=S
: statement to be executed once initially (default pass)

-t, --time
: use time.time() (default on all platforms but Windows)

-c, --clock
: use time.clock() (default on Windows)

-v, --verbose
: print raw timing results; repeat for more digits precision

-h, --help
: print a short usage message and exit

A multi-line statement may be given by specifying each line as a separate statement argument; indented lines are possible by enclosing an argument in quotes and using leading spaces. Multiple -s options are treated similarly.

If -n is not given, a suitable number of loops is calculated by trying successive powers of 10 until the total time is at least 0.2 seconds.

default_timer() measurations can be affected by other programs running on the same machine, so the best thing to do when accurate timing is necessary is to repeat the timing a few times and use the best time. The -r option is good for this; the default of 3 repetitions is probably enough in most cases. On Unix, you can use time.clock() to measure CPU time.

Note: There is a certain baseline overhead associated with executing a pass statement. The code here doesn't try to hide it, but you should be aware of it. The baseline overhead can be measured by invoking the program without arguments, and it might differ between Python versions. Also, to fairly compare older Python versions to Python 2.3, you may want to use Python's -O option (see Optimizations) for the older versions to avoid timing SET_LINENO instructions.

26.6.4 Examples

It is possible to provide a setup statement that is executed only once at the beginning:

```
$ python -m timeit -s 'text = "sample string"; char = "g"'  'char in text'
10000000 loops, best of 3: 0.0877 usec per loop
$ python -m timeit -s 'text = "sample string"; char = "g"'  'text.find(char)'
1000000 loops, best of 3: 0.342 usec per loop
```

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

The same can be done using the Timer class and its methods:

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40193588800002544, 0.3960157959998014, 0.39594301399984033]
```

The following examples show how to time expressions that contain multiple lines. Here we compare the cost of using hasattr() vs. try/except to test for missing and present object attributes:

```
$ python -m timeit 'try:' '  str.__nonzero__' 'except AttributeError:' '  pass'
100000 loops, best of 3: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, "__nonzero__"): pass'
100000 loops, best of 3: 4.26 usec per loop

$ python -m timeit 'try:' '  int.__nonzero__' 'except AttributeError:' '  pass'
1000000 loops, best of 3: 1.43 usec per loop
$ python -m timeit 'if hasattr(int, "__nonzero__"): pass'
100000 loops, best of 3: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__nonzero__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__nonzero__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

To give the timeit module access to functions you define, you can pass a setup parameter which contains an import statement:

```
def test():
    """Stupid test function"""
    L = []
    for i in range(100):
        L.append(i)

if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
import timeit
print(timeit.timeit("test()", setup="from __main__ import test"))
```

26.7 trace — Trace or track Python statement execution

Source code: Lib/trace.py

The trace module allows you to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run. It can be used in another program or from the command line.

26.7.1 Command-Line Usage

The trace module can be invoked from the command line. It can be as simple as

```
python -m trace --count -C . somefile.py ...
```

The above will execute somefile.py and generate annotated listings of all Python modules imported during the execution into the current directory.

--help
: Display usage and exit.

--version
: Display the version of the module and exit.

Main options

At least one of the following options must be specified when invoking trace. The --listfuncs option is mutually exclusive with the --trace and --count options. When --listfuncs is provided, neither --count nor --trace are accepted, and vice versa.

-c, --count
: Produce a set of annotated listing files upon program completion that shows how many times each statement was executed. See also --coverdir, --file and --no-report below.

-t, --trace
: Display lines as they are executed.

-l, --listfuncs
: Display the functions executed by running the program.

-r, --report
: Produce an annotated list from an earlier program run that used the --count and --file option. This does not execute any code.

-T, --trackcalls
: Display the calling relationships exposed by running the program.

Modifiers

-f, --file=<file>
: Name of a file to accumulate counts over several tracing runs. Should be used with the --count option.

-C, --coverdir=<dir>
: Directory where the report files go. The coverage report for package.module is written to file dir/package/module.cover.

-m, --missing
: When generating annotated listings, mark lines which were not executed with >>>>>>.

-s, --summary
: When using --count or --report, write a brief summary to stdout for each file processed.

-R, --no-report
: Do not generate annotated listings. This is useful if you intend to make several runs with --count, and then produce a single set of annotated listings at the end.

-g, --timing
: Prefix each line with the time since the program started. Only used while tracing.

Filters

These options may be repeated multiple times.

--ignore-module=<mod>
: Ignore each of the given module names and its submodules (if it is a package). The argument can be a list of names separated by a comma.

--ignore-dir=<dir>
: Ignore all modules and packages in the named directory and subdirectories. The argument can be a list of directories separated by os.pathsep.

26.7.2 Programmatic Interface

class trace.Trace([count=1[, trace=1[, countfuncs=0[, countcallers=0[, ignoremods=()[, ignoredirs=()[, infile=None[, outfile=None[, timing=False]]]]]]]])
: Create an object to trace execution of a single statement or expression. All parameters are optional. count enables counting of line numbers. trace enables line execution tracing. countfuncs enables listing of the functions called during the run. countcallers enables call relationship tracking. ignoremods is a list of modules or packages to ignore. ignoredirs is a list of directories whose modules or packages should be ignored. infile is the name of the file from which to read stored count information. outfile is the name of the file in which to write updated count information. timing enables a timestamp relative to when tracing was started to be displayed.

  run(cmd)
  : Execute the command and gather statistics from the execution with the current tracing parameters. cmd must be a string or code object, suitable for passing into exec().

  runctx(cmd, globals=None, locals=None)
  : Execute the command and gather statistics from the execution with the current tracing parameters, in the defined global and local environments. If not defined, globals and locals default to empty dictionaries.

runfunc(func, *args, **kwds)
Call func with the given arguments under control of the Trace object with the current tracing parameters.

results()
Return a CoverageResults object that contains the cumulative results of all previous calls to run, runctx and runfunc for the given Trace instance. Does not reset the accumulated trace results.

class trace.CoverageResults
A container for coverage results, created by Trace.results(). Should not be created directly by the user.

update(other)
Merge in data from another CoverageResults object.

write_results([show_missing=True[, summary=False[, coverdir=None]]])
Write coverage results. Set show_missing to show lines that had no hits. Set summary to include in the output the coverage summary per module. coverdir specifies the directory into which the coverage result files will be output. If None, the results for each source file are placed in its directory.

A simple example demonstrating the use of the programmatic interface:

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

CHAPTER

TWENTYSEVEN

SOFTWARE PACKAGING AND DISTRIBUTION

These libraries help you with publishing and installing Python software. While these modules are designed to work in conjunction with the Python Package Index, they can also be used with a local index server, or without any index server at all.

27.1 distutils — Building and installing Python modules

The distutils package provides support for building and installing additional modules into a Python installation. The new modules may be either 100%-pure Python, or may be extension modules written in C, or may be collections of Python packages which include modules coded in both Python and C.

Most Python users will not want to use this module directly, but instead use the cross-version tools maintained by the Python Packaging Authority. In particular, setuptools is an enhanced alternative to distutils that provides:

- support for declaring project dependencies
- additional mechanisms for configuring which files to include in source releases (including plugins for integration with version control systems)
- the ability to declare project "entry points", which can be used as the basis for application plugin systems
- the ability to automatically generate Windows command line executables at installation time rather than needing to prebuild them
- consistent behaviour across all supported Python versions

The recommended pip installer runs all setup.py scripts with setuptools, even if the script itself only imports distutils. Refer to the Python Packaging User Guide for more information.

For the benefits of packaging tool authors and users seeking a deeper understanding of the details of the current packaging and distribution system, the legacy distutils based user documentation and API reference remain available:

- install-index
- distutils-index

27.2 ensurepip — Bootstrapping the pip installer

New in version 2.7.9.

The ensurepip package provides support for bootstrapping the pip installer into an existing Python installation or virtual environment. This bootstrapping approach reflects the fact that pip is an independent project with its own release cycle, and the latest available stable version is bundled with maintenance and feature releases of the CPython reference interpreter.

In most cases, end users of Python shouldn't need to invoke this module directly (as pip should be bootstrapped by default), but it may be needed if installing pip was skipped when installing Python (or when creating a virtual environment) or after explicitly uninstalling pip.

Note: This module does not access the internet. All of the components needed to bootstrap pip are included as internal parts of the package.

See also:

installing-index The end user guide for installing Python packages

PEP 453: Explicit bootstrapping of pip in Python installations The original rationale and specification for this module.

PEP 477: Backport ensurepip (PEP 453) to Python 2.7 The rationale and specification for backporting PEP 453 to Python 2.7.

27.2.1 Command line interface

The command line interface is invoked using the interpreter's -m switch.

The simplest possible invocation is:

```
python -m ensurepip
```

This invocation will install pip if it is not already installed, but otherwise does nothing. To ensure the installed version of pip is at least as recent as the one bundled with ensurepip, pass the --upgrade option:

```
python -m ensurepip --upgrade
```

By default, pip is installed into the current virtual environment (if one is active) or into the system site packages (if there is no active virtual environment). The installation location can be controlled through two additional command line options:

- --root <dir>: Installs pip relative to the given root directory rather than the root of the currently active virtual environment (if any) or the default root for the current Python installation.
- --user: Installs pip into the user site packages directory rather than globally for the current Python installation (this option is not permitted inside an active virtual environment).

By default, the scripts pip, pipX, and pipX.Y will be installed (where X.Y stands for the version of Python used to invoke ensurepip). The scripts installed can be controlled through two additional command line options:

- --altinstall: if an alternate installation is requested, the pip and pipX script will not be installed.
- --no-default-pip: if a non-default installation is request, the pip script will not be installed.

Changed in version 2.7.15: The exit status is non-zero if the command fails.

27.2.2 Module API

ensurepip exposes two functions for programmatic use:

ensurepip.version()
Returns a string specifying the bundled version of pip that will be installed when bootstrapping an environment.

ensurepip.bootstrap(root=None, upgrade=False, user=False, altinstall=False, default_pip=True, verbosity=0)
Bootstraps pip into the current or designated environment.

root specifies an alternative root directory to install relative to. If root is None, then installation uses the default install location for the current environment.

upgrade indicates whether or not to upgrade an existing installation of an earlier version of pip to the bundled version.

user indicates whether to use the user scheme rather than installing globally.

By default, the scripts pip, pipX, and pipX.Y will be installed (where X.Y stands for the current version of Python).

If altinstall is set, then pip and pipX will not be installed.

If default_pip is set to False, then pip will not be installed.

Setting both altinstall and default_pip will trigger ValueError.

verbosity controls the level of output to sys.stdout from the bootstrapping operation.

Note: The bootstrapping process has side effects on both sys.path and os.environ. Invoking the command line interface in a subprocess instead allows these side effects to be avoided.

Note: The bootstrapping process may install additional modules required by pip, but other software should not assume those dependencies will always be present by default (as the dependencies may be removed in a future version of pip).

CHAPTER

TWENTYEIGHT

PYTHON RUNTIME SERVICES

The modules described in this chapter provide a wide range of services related to the Python interpreter and its interaction with its environment. Here's an overview:

28.1 sys — System-specific parameters and functions

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

sys.argv
: The list of command line arguments passed to a Python script. argv[0] is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the -c command line option to the interpreter, argv[0] is set to the string `'-c'`. If no script name was passed to the Python interpreter, argv[0] is the empty string.

  To loop over the standard input, or the list of files given on the command line, see the fileinput module.

sys.byteorder
: An indicator of the native byte order. This will have the value `'big'` on big-endian (most-significant byte first) platforms, and `'little'` on little-endian (least-significant byte first) platforms.

  New in version 2.0.

sys.builtin_module_names
: A tuple of strings giving the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way — modules.keys() only lists the imported modules.)

sys.call_tracing(func, args)
: Call func(*args), while tracing is enabled. The tracing state is saved, and restored afterwards. This is intended to be called from a debugger from a checkpoint, to recursively debug some other code.

sys.copyright
: A string containing the copyright pertaining to the Python interpreter.

sys._clear_type_cache()
: Clear the internal type cache. The type cache is used to speed up attribute and method lookups. Use the function only to drop unnecessary references during reference leak debugging.

  This function should be used for internal and specialized purposes only.

  New in version 2.6.

sys._current_frames()
: Return a dictionary mapping each thread's identifier to the topmost stack frame currently active in that thread at the time the function is called. Note that functions in the traceback module can build the call stack given such a frame.

This is most useful for debugging deadlock: this function does not require the deadlocked threads' cooperation, and such threads' call stacks are frozen for as long as they remain deadlocked. The frame returned for a non-deadlocked thread may bear no relationship to that thread's current activity by the time calling code examines the frame.

This function should be used for internal and specialized purposes only.

New in version 2.5.

sys.dllhandle
Integer specifying the handle of the Python DLL. Availability: Windows.

sys.displayhook(value)
If value is not None, this function prints it to sys.stdout, and saves it in __builtin__._.

sys.displayhook is called on the result of evaluating an expression entered in an interactive Python session. The display of these values can be customized by assigning another one-argument function to sys.displayhook.

sys.dont_write_bytecode
If this is true, Python won't try to write .pyc or .pyo files on the import of source modules. This value is initially set to True or False depending on the -B command line option and the PYTHONDONTWRITEBYTECODE environment variable, but you can set it yourself to control bytecode file generation.

New in version 2.6.

sys.excepthook(type, value, traceback)
This function prints out a given traceback and exception to sys.stderr.

When an exception is raised and uncaught, the interpreter calls sys.excepthook with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt; in a Python program this happens just before the program exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to sys.excepthook.

sys.__displayhook__
sys.__excepthook__
These objects contain the original values of displayhook and excepthook at the start of the program. They are saved so that displayhook and excepthook can be restored in case they happen to get replaced with broken objects.

sys.exc_info()
This function returns a tuple of three values that give information about the exception that is currently being handled. The information returned is specific both to the current thread and to the current stack frame. If the current stack frame is not handling an exception, the information is taken from the calling stack frame, or its caller, and so on until a stack frame is found that is handling an exception. Here, "handling an exception" is defined as "executing or having executed an except clause." For any stack frame, only information about the most recently handled exception is accessible.

If no exception is being handled anywhere on the stack, a tuple containing three None values is returned. Otherwise, the values returned are (type, value, traceback). Their meaning is: type gets the exception type of the exception being handled (a class object); value gets the exception parameter (its associated value or the second argument to raise, which is always a class instance if the exception type is a class object); traceback gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

If exc_clear() is called, this function will return three None values until either another exception is raised in the current thread or the execution stack returns to a frame where another exception is being handled.

Warning: Assigning the traceback return value to a local variable in a function that is handling an exception will cause a circular reference. This will prevent anything referenced by a local variable in the same function or by the traceback from being garbage collected. Since most functions don't need access to the traceback, the best solution is to use something like exctype, value = sys.exc_info()[:2] to extract only the exception type and value. If you do need the traceback, make sure to delete it after use (best done with a try ... finally statement) or to call exc_info() in a function that does not itself handle an exception.

Note: Beginning with Python 2.2, such cycles are automatically reclaimed when garbage collection is enabled and they become unreachable, but it remains more efficient to avoid creating cycles.

sys.exc_clear()
This function clears all information relating to the current or last exception that occurred in the current thread. After calling this function, exc_info() will return three None values until another exception is raised in the current thread or the execution stack returns to a frame where another exception is being handled.

This function is only needed in only a few obscure situations. These include logging and error handling systems that report information on the last or current exception. This function can also be used to try to free resources and trigger object finalization, though no guarantee is made as to what objects will be freed, if any.

New in version 2.3.

sys.exc_type
sys.exc_value
sys.exc_traceback
Deprecated since version 1.5: Use exc_info() instead.

Since they are global variables, they are not specific to the current thread, so their use is not safe in a multi-threaded program. When no exception is being handled, exc_type is set to None and the other two are undefined.

sys.exec_prefix
A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also '/usr/local'. This can be set at build time with the --exec-prefix argument to the configure script. Specifically, all configuration files (e.g. the pyconfig.h header file) are installed in the directory exec_prefix/lib/pythonX.Y/config, and shared library modules are installed in exec_prefix/lib/pythonX.Y/lib-dynload, where X.Y is the version number of Python, for example 2.7.

sys.executable
A string giving the absolute path of the executable binary for the Python interpreter, on systems where this makes sense. If Python is unable to retrieve the real path to its executable, sys.executable will be an empty string or None.

sys.exit([arg])
Exit from Python. This is implemented by raising the SystemExit exception, so cleanup actions specified by finally clauses of try statements are honored, and it is possible to intercept the exit attempt at an outer level.

The optional argument arg can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered "successful termination" and any nonzero value is considered "abnormal termination" by shells and the like. Most systems require it to be in the range 0–127, and produce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped; Unix programs generally use

2 for command line syntax errors and 1 for all other kind of errors. If another type of object is passed, None is equivalent to passing zero, and any other object is printed to stderr and results in an exit code of 1. In particular, sys.exit("some error message") is a quick way to exit a program when an error occurs.

Since exit() ultimately "only" raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted.

sys.exitfunc
: This value is not actually defined by the module, but can be set by the user (or by a program) to specify a clean-up action at program exit. When set, it should be a parameterless function. This function will be called when the interpreter exits. Only one function may be installed in this way; to allow multiple functions which will be called at termination, use the atexit module.

Note: The exit function is not called when the program is killed by a signal, when a Python fatal internal error is detected, or when os._exit() is called.

Deprecated since version 2.4: Use atexit instead.

sys.flags
: The struct sequence flags exposes the status of command line flags. The attributes are read only.

attribute	flag
debug	-d
py3k_warning	-3
division_warning	-Q
division_new	-Qnew
inspect	-i
interactive	-i
optimize	-O or -OO
dont_write_bytecode	-B
no_user_site	-s
no_site	-S
ignore_environment	-E
tabcheck	-t or -tt
verbose	-v
unicode	-U
bytes_warning	-b
hash_randomization	-R

New in version 2.6.

New in version 2.7.3: The hash_randomization attribute.

sys.float_info
: A structseq holding information about the float type. It contains low level information about the precision and internal representation. The values correspond to the various floating-point constants defined in the standard header file float.h for the 'C' programming language; see section 5.2.4.2.2 of the 1999 ISO/IEC C standard [C99], 'Characteristics of floating types', for details.

attribute	float.h macro	explanation
epsilon	DBL_EPSILON	difference between 1 and the least value greater than 1 that is representable as a float
dig	DBL_DIG	maximum number of decimal digits that can be faithfully represented in a float; see below
mant_dig	DBL_MANT_DIG	float precision: the number of base-radix digits in the significand of a float
max	DBL_MAX	maximum representable finite float
max_exp	DBL_MAX_EXP	maximum integer e such that radix**(e-1) is a representable finite float
max_10_exp	DBL_MAX_10_EXP	maximum integer e such that 10**e is in the range of representable finite floats
min	DBL_MIN	minimum positive normalized float
min_exp	DBL_MIN_EXP	minimum integer e such that radix**(e-1) is a normalized float
min_10_exp	DBL_MIN_10_EXP	minimum integer e such that 10**e is a normalized float
radix	FLT_RADIX	radix of exponent representation
rounds	FLT_ROUNDS	integer constant representing the rounding mode used for arithmetic operations. This reflects the value of the system FLT_ROUNDS macro at interpreter startup time. See section 5.2.4.2.2 of the C99 standard for an explanation of the possible values and their meanings.

The attribute sys.float_info.dig needs further explanation. If s is any string representing a decimal number with at most sys.float_info.dig significant digits, then converting s to a float and back again will recover a string representing the same decimal value:

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'    # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back -> same value
'3.14159265358979'
```

But for strings with more than sys.float_info.dig significant digits, this isn't always true:

```
>>> s = '9876543211234567'    # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

New in version 2.6.

sys.float_repr_style
: A string indicating how the repr() function behaves for floats. If the string has value 'short' then for a finite float x, repr(x) aims to produce a short string with the property that float(repr(x)) == x. This is the usual behaviour in Python 2.7 and later. Otherwise, float_repr_style has value 'legacy' and repr(x) behaves in the same way as it did in versions of Python prior to 2.7.

  New in version 2.7.

sys.getcheckinterval()
: Return the interpreter's "check interval"; see setcheckinterval().

  New in version 2.3.

sys.getdefaultencoding()
Return the name of the current default string encoding used by the Unicode implementation.

New in version 2.0.

sys.getdlopenflags()
Return the current value of the flags that are used for dlopen() calls. The flag constants are defined in the dl and DLFCN modules. Availability: Unix.

New in version 2.2.

sys.getfilesystemencoding()
Return the name of the encoding used to convert Unicode filenames into system file names, or None if the system default encoding is used. The result value depends on the operating system:

- On Mac OS X, the encoding is 'utf-8'.
- On Unix, the encoding is the user's preference according to the result of nl_langinfo(CODESET), or None if the nl_langinfo(CODESET) failed.
- On Windows NT+, file names are Unicode natively, so no conversion is performed. getfilesystemencoding() still returns 'mbcs', as this is the encoding that applications should use when they explicitly want to convert Unicode strings to byte strings that are equivalent when used as file names.
- On Windows 9x, the encoding is 'mbcs'.

New in version 2.3.

sys.getrefcount(object)
Return the reference count of the object. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to getrefcount().

sys.getrecursionlimit()
Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by setrecursionlimit().

sys.getsizeof(object[, default])
Return the size of an object in bytes. The object can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.

If given, default will be returned if the object does not provide means to retrieve the size. Otherwise a TypeError will be raised.

getsizeof() calls the object's __sizeof__ method and adds an additional garbage collector overhead if the object is managed by the garbage collector.

New in version 2.6.

sys._getframe([depth])
Return a frame object from the call stack. If optional integer depth is given, return the frame object that many calls below the top of the stack. If that is deeper than the call stack, ValueError is raised. The default for depth is zero, returning the frame at the top of the call stack.

CPython implementation detail: This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

sys.getprofile()
Get the profiler function as set by setprofile().

New in version 2.6.

sys.gettrace()
Get the trace function as set by settrace().

CPython implementation detail: The gettrace() function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

New in version 2.6.

sys.getwindowsversion()
Return a named tuple describing the Windows version currently running. The named elements are major, minor, build, platform, service_pack, service_pack_minor, service_pack_major, suite_mask, and product_type. service_pack contains a string while all other values are integers. The components can also be accessed by name, so sys.getwindowsversion()[0] is equivalent to sys.getwindowsversion().major. For compatibility with prior versions, only the first 5 elements are retrievable by indexing.

platform may be one of the following values:

Constant	Platform
0 (VER_PLATFORM_WIN32s)	Win32s on Windows 3.1
1 (VER_PLATFORM_WIN32_WINDOWS)	Windows 95/98/ME
2 (VER_PLATFORM_WIN32_NT)	Windows NT/2000/XP/x64
3 (VER_PLATFORM_WIN32_CE)	Windows CE

product_type may be one of the following values:

Constant	Meaning
1 (VER_NT_WORKSTATION)	The system is a workstation.
2 (VER_NT_DOMAIN_CONTROLLER)	The system is a domain controller.
3 (VER_NT_SERVER)	The system is a server, but not a domain controller.

This function wraps the Win32 GetVersionEx() function; see the Microsoft documentation on OSVERSIONINFOEX() for more information about these fields.

Availability: Windows.

New in version 2.3.

Changed in version 2.7: Changed to a named tuple and added service_pack_minor, service_pack_major, suite_mask, and product_type.

sys.hexversion
The version number encoded as a single integer. This is guaranteed to increase with each version, including proper support for non-production releases. For example, to test that the Python interpreter is at least version 1.5.2, use:

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

This is called hexversion since it only really looks meaningful when viewed as the result of passing it to the built-in hex() function. The version_info value may be used for a more human-friendly encoding of the same information.

The hexversion is a 32-bit number with the following layout:

Bits (big endian order)	Meaning
1-8	PY_MAJOR_VERSION (the 2 in 2.1.0a3)
9-16	PY_MINOR_VERSION (the 1 in 2.1.0a3)
17-24	PY_MICRO_VERSION (the 0 in 2.1.0a3)
25-28	PY_RELEASE_LEVEL (0xA for alpha, 0xB for beta, 0xC for release candidate and 0xF for final)
29-32	PY_RELEASE_SERIAL (the 3 in 2.1.0a3, zero for final releases)

Thus 2.1.0a3 is hexversion 0x020100a3.

New in version 1.5.2.

sys.long_info
A struct sequence that holds information about Python's internal representation of integers. The attributes are read only.

Attribute	Explanation
bits_per_digit	number of bits held in each digit. Python integers are stored internally in base 2**long_info.bits_per_digit
sizeof_digit	size in bytes of the C type used to represent a digit

New in version 2.7.

sys.last_type
sys.last_value
sys.last_traceback
These three variables are not always defined; they are set when an exception is not handled and the interpreter prints an error message and a stack traceback. Their intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is import pdb; pdb.pm() to enter the post-mortem debugger; see chapter pdb — The Python Debugger for more information.)

The meaning of the variables is the same as that of the return values from exc_info() above. (Since there is only one interactive thread, thread-safety is not a concern for these variables, unlike for exc_type etc.)

sys.maxint
The largest positive integer supported by Python's regular integer type. This is at least 2**31-1. The largest negative integer is -maxint-1 — the asymmetry results from the use of 2's complement binary arithmetic.

sys.maxsize
The largest positive integer supported by the platform's Py_ssize_t type, and thus the maximum size lists, strings, dicts, and many other containers can have.

sys.maxunicode
An integer giving the largest supported code point for a Unicode character. The value of this depends on the configuration option that specifies whether Unicode characters are stored as UCS-2 or UCS-4.

sys.meta_path
A list of finder objects that have their find_module() methods called to see if one of the objects can find the module to be imported. The find_module() method is called at least with the absolute name of the module being imported. If the module to be imported is contained in package then the parent package's __path__ attribute is passed in as a second argument. The method returns None if the module cannot be found, else returns a loader.

sys.meta_path is searched before any implicit default finders or sys.path.

See PEP 302 for the original specification.

sys.modules
: This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. Note that removing a module from this dictionary is not the same as calling reload() on the corresponding module object.

sys.path
: A list of strings that specifies the search path for modules. Initialized from the environment variable PYTHONPATH, plus an installation-dependent default.

  As initialized upon program startup, the first item of this list, path[0], is the directory containing the script that was used to invoke the Python interpreter. If the script directory is not available (e.g. if the interpreter is invoked interactively or if the script is read from standard input), path[0] is the empty string, which directs Python to search modules in the current directory first. Notice that the script directory is inserted before the entries inserted as a result of PYTHONPATH.

  A program is free to modify this list for its own purposes.

  Changed in version 2.3: Unicode strings are no longer ignored.

  See also:

  Module site This describes how to use .pth files to extend sys.path.

sys.path_hooks
: A list of callables that take a path argument to try to create a finder for the path. If a finder can be created, it is to be returned by the callable, else raise ImportError.

  Originally specified in PEP 302.

sys.path_importer_cache
: A dictionary acting as a cache for finder objects. The keys are paths that have been passed to sys.path_hooks and the values are the finders that are found. If a path is a valid file system path but no explicit finder is found on sys.path_hooks then None is stored to represent the implicit default finder should be used. If the path is not an existing path then imp.NullImporter is set.

  Originally specified in PEP 302.

sys.platform
: This string contains a platform identifier that can be used to append platform-specific components to sys.path, for instance.

  For most Unix systems, this is the lowercased OS name as returned by uname -s with the first part of the version as returned by uname -r appended, e.g. `'sunos5'`, at the time when Python was built. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom:

```
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
```

  Changed in version 2.7.3: Since lots of code check for sys.platform == `'linux2'`, and there is no essential change between Linux 2.x and 3.x, sys.platform is always set to `'linux2'`, even on Linux 3.x. In Python 3.3 and later, the value will always be set to `'linux'`, so it is recommended to always use the startswith idiom presented above.

  For other systems, the values are:

System	platform value
Linux (2.x and 3.x)	'linux2'
Windows	'win32'
Windows/Cygwin	'cygwin'
Mac OS X	'darwin'
OS/2	'os2'
OS/2 EMX	'os2emx'
RiscOS	'riscos'
AtheOS	'atheos'

See also:

os.name has a coarser granularity. os.uname() gives system-dependent version information.

The platform module provides detailed checks for the system's identity.

sys.prefix
: A string giving the site-specific directory prefix where the platform independent Python files are installed; by default, this is the string '/usr/local'. This can be set at build time with the --prefix argument to the configure script. The main collection of Python library modules is installed in the directory prefix/lib/pythonX.Y while the platform independent header files (all except pyconfig.h) are stored in prefix/include/pythonX.Y, where X.Y is the version number of Python, for example 2.7.

sys.ps1
sys.ps2
: Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are '>>> ' and '... '. If a non-string object is assigned to either variable, its str() is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

sys.py3kwarning
: Bool containing the status of the Python 3 warning flag. It's True when Python is started with the -3 option. (This should be considered read-only; setting it to a different value doesn't have an effect on Python 3 warnings.)

  New in version 2.6.

sys.setcheckinterval(interval)
: Set the interpreter's "check interval". This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is 100, meaning the check is performed every 100 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value <= 0 checks every virtual instruction, maximizing responsiveness as well as overhead.

sys.setdefaultencoding(name)
: Set the current default string encoding used by the Unicode implementation. If name does not match any available encoding, LookupError is raised. This function is only intended to be used by the site module implementation and, where needed, by sitecustomize. Once used by the site module, it is removed from the sys module's namespace.

  New in version 2.0.

sys.setdlopenflags(n)
: Set the flags used by the interpreter for dlopen() calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as sys.setdlopenflags(0). To share symbols across extension modules, call as sys.setdlopenflags(dl.RTLD_NOW | dl.RTLD_GLOBAL). Symbolic names for the flag modules can be either found in the

dl module, or in the DLFCN module. If DLFCN is not available, it can be generated from /usr/include/dlfcn.h using the h2py script. Availability: Unix.

New in version 2.2.

sys.setprofile(profilefunc)
Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter The Python Profilers for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see settrace()), but it is called with different events, for example it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return None.

Profile functions should have three arguments: frame, event, and arg. frame is the current stack frame. event is a string: 'call', 'return', 'c_call', 'c_return', or 'c_exception'. arg depends on the event type.

The events have the following meaning:

'call' A function is called (or some other code block entered). The profile function is called; arg is None.

'return' A function (or other code block) is about to return. The profile function is called; arg is the value that will be returned, or None if the event is caused by an exception being raised.

'c_call' A C function is about to be called. This may be an extension function or a built-in. arg is the C function object.

'c_return' A C function has returned. arg is the C function object.

'c_exception' A C function has raised an exception. arg is the C function object.

sys.setrecursionlimit(limit)
Set the maximum depth of the Python interpreter stack to limit. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

The highest possible limit is platform-dependent. A user may need to set the limit higher when she has a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

sys.settrace(tracefunc)
Set the system's trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must be registered using settrace() for each thread being debugged.

Trace functions should have three arguments: frame, event, and arg. frame is the current stack frame. event is a string: 'call', 'line', 'return' or 'exception'. arg depends on the event type.

The trace function is invoked (with event set to 'call') whenever a new local scope is entered; it should return a reference to a local trace function to be used that scope, or None if the scope shouldn't be traced.

The local trace function should return a reference to itself (or to another function for further tracing in that scope), or None to turn off tracing in that scope.

The events have the following meaning:

'call' A function is called (or some other code block entered). The global trace function is called; arg is None; the return value specifies the local trace function.

'line' The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; arg is None; the return value specifies the new local trace function. See Objects/lnotab_notes.txt for a detailed explanation of how this works.

'return' A function (or other code block) is about to return. The local trace function is called; arg is the value that will be returned, or None if the event is caused by an exception being raised. The trace function's return value is ignored.

'exception' An exception has occurred. The local trace function is called; arg is a tuple (exception, value, traceback); the return value specifies the new local trace function.

Note that as an exception is propagated down the chain of callers, an 'exception' event is generated at each level.

For more information on code and frame objects, refer to types.

CPython implementation detail: The settrace() function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

sys.settscdump(on_flag)
Activate dumping of VM measurements using the Pentium timestamp counter, if on_flag is true. Deactivate these dumps if on_flag is off. The function is available only if Python was compiled with --with-tsc. To understand the output of this dump, read Python/ceval.c in the Python sources.

New in version 2.4.

CPython implementation detail: This function is intimately bound to CPython implementation details and thus not likely to be implemented elsewhere.

sys.stdin
sys.stdout
sys.stderr
File objects corresponding to the interpreter's standard input, output and error streams. stdin is used for all interpreter input except for scripts but including calls to input() and raw_input(). stdout is used for the output of print and expression statements and for the prompts of input() and raw_input(). The interpreter's own prompts and (almost all of) its error messages go to stderr. stdout and stderr needn't be built-in file objects: any object is acceptable as long as it has a write() method that takes a string argument. (Changing these objects doesn't affect the standard I/O streams of processes executed by os.popen(), os.system() or the exec*() family of functions in the os module.)

sys.__stdin__
sys.__stdout__
sys.__stderr__
These objects contain the original values of stdin, stderr and stdout at the start of the program. They are used during finalization, and could be useful to print to the actual standard stream no matter if the sys.std* object has been redirected.

It can also be used to restore the actual files to known working file objects in case they have been overwritten with a broken object. However, the preferred way to do this is to explicitly save the previous stream before replacing it, and restore the saved object.

sys.subversion
A triple (repo, branch, version) representing the Subversion information of the Python interpreter. repo is the name of the repository, 'CPython'. branch is a string of one of the forms 'trunk', 'branches/name' or 'tags/name'. version is the output of svnversion, if the interpreter was built from a Subversion checkout; it contains the revision number (range) and possibly a trailing 'M' if there were local modifications. If the tree was exported (or svnversion was not available), it is the revision of Include/patchlevel.h if the branch is a tag. Otherwise, it is None.

New in version 2.5.

Note: Python is now developed using Mercurial. In recent Python 2.7 bugfix releases, subversion therefore contains placeholder information. It is removed in Python 3.3.

---

sys.tracebacklimit
When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is 1000. When set to 0 or less, all traceback information is suppressed and only the exception type and value are printed.

sys.version
A string containing the version number of the Python interpreter plus additional information on the build number and compiler used. This string is displayed when the interactive interpreter is started. Do not extract version information out of it, rather, use version_info and the functions provided by the platform module.

sys.api_version
The C API version for this interpreter. Programmers may find this useful when debugging version conflicts between Python and extension modules.

New in version 2.3.

sys.version_info
A tuple containing the five components of the version number: major, minor, micro, releaselevel, and serial. All values except releaselevel are integers; the release level is 'alpha', 'beta', 'candidate', or 'final'. The version_info value corresponding to the Python version 2.0 is (2, 0, 0, 'final', 0). The components can also be accessed by name, so sys.version_info[0] is equivalent to sys.version_info.major and so on.

New in version 2.0.

Changed in version 2.7: Added named component attributes

sys.warnoptions
This is an implementation detail of the warnings framework; do not modify this value. Refer to the warnings module for more information on the warnings framework.

sys.winver
The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the first three characters of version. It is provided in the sys module for informational purposes; modifying this value has no effect on the registry keys used by Python. Availability: Windows.

Citations

28.2 sysconfig — Provide access to Python's configuration information

New in version 2.7.

Source code: Lib/sysconfig.py

The sysconfig module provides access to Python's configuration information like the list of installation paths and the configuration variables relevant for the current platform.

28.2.1 Configuration variables

A Python distribution contains a Makefile and a pyconfig.h header file that are necessary to build both the Python binary itself and third-party C extensions compiled using distutils.

sysconfig puts all variables found in these files in a dictionary that can be accessed using get_config_vars() or get_config_var().

Notice that on Windows, it's a much smaller set.

sysconfig.get_config_vars(*args)
: With no arguments, return a dictionary of all configuration variables relevant for the current platform.

  With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

  For each argument, if the value is not found, return None.

sysconfig.get_config_var(name)
: Return the value of a single variable name. Equivalent to get_config_vars().get(name).

  If name is not found, return None.

Example of usage:

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

28.2.2 Installation paths

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in sysconfig under unique identifiers based on the value returned by os.name.

Every new component that is installed using distutils or a Distutils-based system will follow the same scheme to copy its file in the right places.

Python currently supports seven schemes:

- posix_prefix: scheme for Posix platforms like Linux or Mac OS X. This is the default scheme used when Python or a component is installed.
- posix_home: scheme for Posix platforms used when a home option is used upon installation. This scheme is used when a component is installed through Distutils with a specific home prefix.
- posix_user: scheme for Posix platforms used when a component is installed through Distutils and the user option is used. This scheme defines paths located under the user home directory.
- nt: scheme for NT platforms like Windows.
- nt_user: scheme for NT platforms, when the user option is used.
- os2: scheme for OS/2 platforms.
- os2_home: scheme for OS/2 platforms, when the user option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths:

- stdlib: directory containing the standard Python library files that are not platform-specific.
- platstdlib: directory containing the standard Python library files that are platform-specific.
- platlib: directory for site-specific, platform-specific files.
- purelib: directory for site-specific, non-platform-specific files.
- include: directory for non-platform-specific header files.
- platinclude: directory for platform-specific header files.
- scripts: directory for script files.
- data: directory for data files.

sysconfig provides some functions to determine these paths.

sysconfig.get_scheme_names()
Return a tuple containing all schemes currently supported in sysconfig.

sysconfig.get_path_names()
Return a tuple containing all path names currently supported in sysconfig.

sysconfig.get_path(name[, scheme[, vars[, expand]]])
Return an installation path corresponding to the path name, from the install scheme named scheme.

name has to be a value from the list returned by get_path_names().

sysconfig stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the stdlib path for the nt scheme is: {base}/Lib.

get_path() will use the variables returned by get_config_vars() to expand the path. All variables have default values for each platform so one may call this function and get the default value.

If scheme is provided, it must be a value from the list returned by get_scheme_names(). Otherwise, the default scheme for the current platform is used.

If vars is provided, it must be a dictionary of variables that will update the dictionary return by get_config_vars().

If expand is set to False, the path will not be expanded using the variables.

If name is not found, return None.

sysconfig.get_paths([scheme[, vars[, expand]]])
Return a dictionary containing all installation paths corresponding to an installation scheme. See get_path() for more information.

If scheme is not provided, will use the default scheme for the current platform.

If vars is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If expand is set to false, the paths will not be expanded.

If scheme is not an existing scheme, get_paths() will raise a KeyError.

28.2.3 Other functions

sysconfig.get_python_version()
Return the MAJOR.MINOR Python version number as a string. Similar to sys.version[:3].

sysconfig.get_platform()
Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by os.uname()), although the exact information included depends on the OS; e.g. for IRIX the architecture isn't particularly important (IRIX only runs on SGI hardware), but for Linux the kernel version isn't particularly important.

Examples of returned values:

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u
- irix-5.3
- irix64-6.2

Windows will return one of:

- win-amd64 (64bit Windows on AMD64, aka x86_64, Intel64, and EM64T)
- win-ia64 (64bit Windows on Itanium)
- win32 (all others - specifically, sys.platform is returned)

Mac OS X can return:

- macosx-10.6-ppc
- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat

For other non-POSIX platforms, currently just returns sys.platform.

sysconfig.is_python_build()
Return True if the current Python installation was built from source.

sysconfig.parse_config_h(fp[, vars])
Parse a config.h-style file.

fp is a file-like object pointing to the config.h-like file.

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

sysconfig.get_config_h_filename()
Return the path of pyconfig.h.

sysconfig.get_makefile_filename()
Return the path of Makefile.

28.3 __builtin__ — Built-in objects

This module provides direct access to all 'built-in' identifiers of Python; for example, __builtin__.open is the full name for the built-in function open(). See Built-in Functions and Built-in Constants for documentation.

This module is not normally accessed explicitly by most applications, but can be useful in modules that provide objects with the same name as a built-in value, but in which the built-in of that name is also needed. For example, in a module that wants to implement an open() function that wraps the built-in open(), this module can be used directly:

```
import __builtin__

def open(path):
    f = __builtin__.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

CPython implementation detail: Most modules have the name __builtins__ (note the 's') made available as part of their globals. The value of __builtins__ is normally either this module or the value of this modules's __dict__ attribute. Since this is an implementation detail, it may not be used by alternate implementations of Python.

28.4 future_builtins — Python 3 builtins

New in version 2.6.

This module provides functions that exist in 2.x, but have different behavior in Python 3, so they cannot be put into the 2.x builtins namespace.

Instead, if you want to write code compatible with Python 3 builtins, import them from this module, like this:

```
from future_builtins import map, filter

... code using Python 3-style map and filter ...
```

The 2to3 tool that ports Python 2 code to Python 3 will recognize this usage and leave the new builtins alone.

Note: The Python 3 print() function is already in the builtins, but cannot be accessed from Python 2 code unless you use the appropriate future statement:

```
from __future__ import print_function
```

Available builtins are:

future_builtins.ascii(object)
: Returns the same as repr(). In Python 3, repr() will return printable Unicode characters unescaped, while ascii() will always backslash-escape them. Using future_builtins.ascii() instead of repr() in 2.6 code makes it clear that you need a pure ASCII return value.

future_builtins.filter(function, iterable)
Works like itertools.ifilter().

future_builtins.hex(object)
Works like the built-in hex(), but instead of __hex__() it will use the __index__() method on its argument to get an integer that is then converted to hexadecimal.

future_builtins.map(function, iterable, ...)
Works like itertools.imap().

Note: In Python 3, map() does not accept None for the function argument.

---

future_builtins.oct(object)
Works like the built-in oct(), but instead of __oct__() it will use the __index__() method on its argument to get an integer that is then converted to octal.

future_builtins.zip(*iterables)
Works like itertools.izip().

28.5 __main__ — Top-level script environment

This module represents the (otherwise anonymous) scope in which the interpreter's main program executes — commands read either from standard input, from a script file, or from an interactive prompt. It is this environment in which the idiomatic "conditional script" stanza causes a script to run:

```
if __name__ == "__main__":
    main()
```

28.6 warnings — Warning control

New in version 2.1.

Source code: Lib/warnings.py

Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn't warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

Python programmers issue warnings by calling the warn() function defined in this module. (C programmers use PyErr_WarnEx(); see exceptionhandling for details).

Warning messages are normally written to sys.stderr, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the warning category (see below), the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

There are two stages in warning control: first, each time a warning is issued, a determination is made whether a message should be issued or not; next, if a message is to be issued, it is formatted and printed using a user-settable hook.

The determination whether to issue a warning message is controlled by the warning filter, which is a sequence of matching rules and actions. Rules can be added to the filter by calling filterwarnings() and reset to its default state by calling resetwarnings().

The printing of warning messages is done by calling showwarning(), which may be overridden; the default implementation of this function formats the message by calling formatwarning(), which is also available for use by custom implementations.

See also:

logging.captureWarnings() allows you to handle all warnings with the standard logging infrastructure.

28.6.1 Warning Categories

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings. The following warnings category classes are currently defined:

Class	Description
Warning	This is the base class of all warning category classes. It is a subclass of Exception.
UserWarning	The default category for warn().
DeprecationWarning	Base category for warnings about deprecated features (ignored by default).
SyntaxWarning	Base category for warnings about dubious syntactic features.
RuntimeWarning	Base category for warnings about dubious runtime features.
FutureWarning	Base category for warnings about constructs that will change semantically in the future.
PendingDeprecationWarning	Base category for warnings about features that will be deprecated in the future (ignored by default).
ImportWarning	Base category for warnings triggered during the process of importing a module (ignored by default).
UnicodeWarning	Base category for warnings related to Unicode.

While these are technically built-in exceptions, they are documented here, because conceptually they belong to the warnings mechanism.

User code can define additional warning categories by subclassing one of the standard warning categories. A warning category must always be a subclass of the Warning class.

Changed in version 2.7: DeprecationWarning is ignored by default.

28.6.2 The Warnings Filter

The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications; any specific warning is matched against each filter specification in the list in turn until a match is found; the match determines the disposition of the match. Each entry is a tuple of the form (action, message, category, module, lineno), where:

- action is one of the following strings:

Value	Disposition
"error"	turn matching warnings into exceptions
"ignore"	never print matching warnings
"always"	always print matching warnings
"default"	print the first occurrence of matching warnings for each location where the warning is issued
"module"	print the first occurrence of matching warnings for each module where the warning is issued
"once"	print only the first occurrence of matching warnings, regardless of location

- message is a string containing a regular expression that the start of the warning message must match. The expression is compiled to always be case-insensitive.
- category is a class (a subclass of Warning) of which the warning category must be a subclass in order to match.
- module is a string containing a regular expression that the module name must match. The expression is compiled to be case-sensitive.
- lineno is an integer that the line number where the warning occurred must match, or 0 to match all line numbers.

Since the Warning class is derived from the built-in Exception class, to turn a warning into an error we simply raise category(message).

The warnings filter is initialized by -W options passed to the Python interpreter command line. The interpreter saves the arguments for all -W options without interpretation in sys.warnoptions; the warnings module parses these when it is first imported (invalid options are ignored, after printing a message to sys.stderr).

Default Warning Filters

By default, Python installs several warning filters, which can be overridden by the command-line options passed to -W and calls to filterwarnings().

- DeprecationWarning and PendingDeprecationWarning, and ImportWarning are ignored.
- BytesWarning is ignored unless the -b option is given once or twice; in this case this warning is either printed (-b) or turned into an exception (-bb).

28.6.3 Temporarily Suppressing Warnings

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning, then it is possible to suppress the warning using the catch_warnings context manager:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

While within the context manager all warnings will simply be ignored. This allows you to use known-deprecated code without having to see the warning while not suppressing the warning for other code that might not be aware of its use of deprecated code. Note: this can only be guaranteed in a single-threaded

application. If two or more threads use the catch_warnings context manager at the same time, the behavior is undefined.

28.6.4 Testing Warnings

To test warnings raised by code, use the catch_warnings context manager. With it you can temporarily mutate the warnings filter to facilitate your testing. For instance, do the following to capture all raised warnings to check:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

One can also cause all warnings to be exceptions by using error instead of always. One thing to be aware of is that if a warning has already been raised because of a once/default rule, then no matter what filters are set the warning will not be seen again unless the warnings registry related to the warning has been cleared.

Once the context manager exits, the warnings filter is restored to its state when the context was entered. This prevents tests from changing the warnings filter in unexpected ways between tests and leading to indeterminate test results. The showwarning() function in the module is also restored to its original value. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the catch_warnings context manager at the same time, the behavior is undefined.

When testing multiple operations that raise the same kind of warning, it is important to test them in a manner that confirms each operation is raising a new warning (e.g. set warnings to be raised as exceptions and check the operations raise exceptions, check that the length of the warning list continues to increase after each operation, or else delete the previous entries from the warnings list before each new operation).

28.6.5 Updating Code For New Versions of Python

Warnings that are only of interest to the developer are ignored by default. As such you should make sure to test your code with typically ignored warnings made visible. You can do this from the command-line by passing -Wd to the interpreter (this is shorthand for -W default). This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you simply change what argument is passed to -W, e.g. -W error. See the -W flag for more details on what is possible.

To programmatically do the same as -Wd, use:

```
warnings.simplefilter('default')
```

Make sure to execute this code as soon as possible. This prevents the registering of what warnings have been raised from unexpectedly influencing how future warnings are treated.

Having certain warnings ignored by default is done to prevent a user from seeing warnings that are only of interest to the developer. As you do not necessarily have control over what interpreter a user uses to run their code, it is possible that a new version of Python will be released between your release cycles. The new interpreter release could trigger new warnings in your code that were not there in an older interpreter, e.g. DeprecationWarning for a module that you are using. While you as a developer want to be notified that your code is using a deprecated module, to a user this information is essentially noise and provides no benefit to them.

28.6.6 Available Functions

warnings.warn(message[, category[, stacklevel]])
Issue a warning, or maybe ignore it or raise an exception. The category argument, if given, must be a warning category class (see above); it defaults to UserWarning. Alternatively message can be a Warning instance, in which case category will be ignored and message.__class__ will be used. In this case the message text will be str(message). This function raises an exception if the particular warning issued is changed into an error by the warnings filter see above. The stacklevel argument can be used by wrapper functions written in Python, like this:

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to deprecation()'s caller, rather than to the source of deprecation() itself (since the latter would defeat the purpose of the warning message).

warnings.warn_explicit(message, category, filename, lineno[, module[, registry[, module_globals]]])
This is a low-level interface to the functionality of warn(), passing in explicitly the message, category, filename and line number, and optionally the module name and the registry (which should be the __warningregistry__ dictionary of the module). The module name defaults to the filename with .py stripped; if no registry is passed, the warning is never suppressed. message must be a string and category a subclass of Warning or message may be a Warning instance, in which case category will be ignored.

module_globals, if supplied, should be the global namespace in use by the code for which the warning is issued. (This argument is used to support displaying source for modules found in zipfiles or other non-filesystem import sources).

Changed in version 2.5: Added the module_globals parameter.

warnings.warnpy3k(message[, category[, stacklevel]])
Issue a warning related to Python 3.x deprecation. Warnings are only shown when Python is started with the -3 option. Like warn() message must be a string and category a subclass of Warning. warnpy3k() is using DeprecationWarning as default warning class.

New in version 2.6.

warnings.showwarning(message, category, filename, lineno[, file[, line]])
Write a warning to a file. The default implementation calls formatwarning(message, category, filename, lineno, line) and writes the resulting string to file, which defaults to sys.stderr. You may replace this function with an alternative implementation by assigning to warnings.showwarning. line is a line of source code to be included in the warning message; if line is not supplied, showwarning() will try to read the line specified by filename and lineno.

Changed in version 2.7: The line argument is required to be supported.

warnings.formatwarning(message, category, filename, lineno[, line])
Format a warning the standard way. This returns a string which may contain embedded newlines and

ends in a newline. line is a line of source code to be included in the warning message; if line is not supplied, formatwarning() will try to read the line specified by filename and lineno.

Changed in version 2.6: Added the line argument.

warnings.filterwarnings(action[, message[, category[, module[, lineno[, append]]]]])
Insert an entry into the list of warnings filter specifications. The entry is inserted at the front by default; if append is true, it is inserted at the end. This checks the types of the arguments, compiles the message and module regular expressions, and inserts them as a tuple in the list of warnings filters. Entries closer to the front of the list override entries later in the list, if both match a particular warning. Omitted arguments default to a value that matches everything.

warnings.simplefilter(action[, category[, lineno[, append]]])
Insert a simple entry into the list of warnings filter specifications. The meaning of the function parameters is as for filterwarnings(), but regular expressions are not needed as the filter inserted always matches any message in any module as long as the category and line number match.

warnings.resetwarnings()
Reset the warnings filter. This discards the effect of all previous calls to filterwarnings(), including that of the -W command line options and calls to simplefilter().

28.6.7 Available Context Managers

class warnings.catch_warnings([*, record=False, module=None])
A context manager that copies and, upon exit, restores the warnings filter and the showwarning() function. If the record argument is False (the default) the context manager returns None on entry. If record is True, a list is returned that is progressively populated with objects as seen by a custom showwarning() function (which also suppresses output to sys.stdout). Each object in the list has attributes with the same names as the arguments to showwarning().

The module argument takes a module that will be used instead of the module returned when you import warnings whose filter will be protected. This argument exists primarily for testing the warnings module itself.

Note: The catch_warnings manager works by replacing and then later restoring the module's showwarning() function and internal list of filter specifications. This means the context manager is modifying global state and therefore is not thread-safe.

Note: In Python 3, the arguments to the constructor for catch_warnings are keyword-only arguments.

New in version 2.6.

28.7 contextlib — Utilities for with-statement contexts

New in version 2.5.

Source code: Lib/contextlib.py

This module provides utilities for common tasks involving the with statement. For more information see also Context Manager Types and context-managers.

Functions provided:

contextlib.contextmanager(func)

This function is a decorator that can be used to define a factory function for with statement context managers, without needing to create a class or separate __enter__() and __exit__() methods.

While many objects natively support use in with statements, sometimes a resource needs to be managed that isn't a context manager in its own right, and doesn't implement a close() method for use with contextlib.closing

An abstract example would be the following to ensure correct resource management:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)

>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

The function being decorated must return a generator-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the with statement's as clause, if any.

At the point where the generator yields, the block nested in the with statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a try...except...finally statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the generator must reraise that exception. Otherwise the generator context manager will indicate to the with statement that the exception has been handled, and execution will resume with the statement immediately following the with statement.

contextlib.nested(mgr1[, mgr2[, ...]])

Combine multiple context managers into a single nested context manager.

This function has been deprecated in favour of the multiple manager form of the with statement.

The one advantage of this function over the multiple manager form of the with statement is that argument unpacking allows it to be used with a variable number of context managers as follows:

```
from contextlib import nested

with nested(*managers):
    do_something()
```

Note that if the __exit__() method of one of the nested context managers indicates an exception should be suppressed, no exception information will be passed to any remaining outer context managers. Similarly, if the __exit__() method of one of the nested managers raises an exception, any previous exception state will be lost; the new exception will be passed to the __exit__() methods of any remaining outer context managers. In general, __exit__() methods should avoid raising exceptions, and in particular they should not re-raise a passed-in exception.

This function has two major quirks that have led to it being deprecated. Firstly, as the context managers are all constructed before the function is invoked, the __new__() and __init__() methods of the inner context managers are not actually covered by the scope of the outer context managers. That means, for example, that using nested() to open two files is a programming error as the first file will not be closed promptly if an exception is thrown when opening the second file.

Secondly, if the __enter__() method of one of the inner context managers raises an exception that is caught and suppressed by the __exit__() method of one of the outer context managers, this construct will raise RuntimeError rather than skipping the body of the with statement.

Developers that need to support nesting of a variable number of context managers can either use the warnings module to suppress the DeprecationWarning raised by this function or else use this function as a model for an application specific implementation.

Deprecated since version 2.7: The with-statement now supports this functionality directly (without the confusing error prone quirks).

contextlib.closing(thing)

Return a context manager that closes thing upon completion of the block. This is basically equivalent to:

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

And lets you write code like this:

```
from contextlib import closing
import urllib

with closing(urllib.urlopen('http://www.python.org')) as page:
    for line in page:
        print line
```

without needing to explicitly close page. Even if an error occurs, page.close() will be called when the with block is exited.

See also:

PEP 343 - The "with" statement The specification, background, and examples for the Python with statement.

28.8 abc — Abstract Base Classes

New in version 2.6.

Source code: Lib/abc.py

This module provides the infrastructure for defining abstract base classes (ABCs) in Python, as outlined in PEP 3119; see the PEP for why this was added to Python. (See also PEP 3141 and the numbers module regarding a type hierarchy for numbers based on ABCs.)

The collections module has some concrete classes that derive from ABCs; these can, of course, be further derived. In addition, the collections module has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, if it is hashable or if it is a mapping.

This module provides the following class:

class abc.ABCMeta
Metaclass for defining Abstract Base Classes (ABCs).

Use this metaclass to create an ABC. An ABC can be subclassed directly, and then acts as a mix-in class. You can also register unrelated concrete classes (even built-in classes) and unrelated ABCs as "virtual subclasses" – these and their descendants will be considered subclasses of the registering ABC by the built-in issubclass() function, but the registering ABC won't show up in their MRO (Method Resolution Order) nor will method implementations defined by the registering ABC be callable (not even via super()).[1]

Classes created with a metaclass of ABCMeta have the following method:

register(subclass)
Register subclass as a "virtual subclass" of this ABC. For example:

```
from abc import ABCMeta

class MyABC:
    __metaclass__ = ABCMeta

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

You can also override this method in an abstract base class:

__subclasshook__(subclass)
(Must be defined as a class method.)

Check whether subclass is considered a subclass of this ABC. This means that you can customize the behavior of issubclass further without the need to call register() on every class you want to consider a subclass of the ABC. (This class method is called from the __subclasscheck__() method of the ABC.)

This method should return True, False or NotImplemented. If it returns True, the subclass is considered a subclass of this ABC. If it returns False, the subclass is not considered a subclass of this ABC, even if it would normally be one. If it returns NotImplemented, the subclass check is continued with the usual mechanism.

For a demonstration of these concepts, look at this example ABC definition:

```
class Foo(object):
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable:
    __metaclass__ = ABCMeta
```

(continues on next page)

[1] C++ programmers should note that Python's virtual base class concept is not the same as C++'s.

(continued from previous page)

```
    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented

MyIterable.register(Foo)
```

The ABC MyIterable defines the standard iterable method, `__iter__()`, as an abstract method. The implementation given here can still be called from subclasses. The get_iterator() method is also part of the MyIterable abstract base class, but it does not have to be overridden in non-abstract derived classes.

The `__subclasshook__()` class method defined here says that any class that has an `__iter__()` method in its `__dict__` (or in that of one of its base classes, accessed via the `__mro__` list) is considered a MyIterable too.

Finally, the last line makes Foo a virtual subclass of MyIterable, even though it does not define an `__iter__()` method (it uses the old-style iterable protocol, defined in terms of `__len__()` and `__getitem__()`). Note that this will not make get_iterator available as a method of Foo, so it is provided separately.

It also provides the following decorators:

abc.abstractmethod(function)
: A decorator indicating abstract methods.

  Using this decorator requires that the class's metaclass is ABCMeta or is derived from it. A class that has a metaclass derived from ABCMeta cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract methods can be called using any of the normal 'super' call mechanisms.

  Dynamically adding abstract methods to a class, or attempting to modify the abstraction status of a method or class once it is created, are not supported. The abstractmethod() only affects subclasses derived using regular inheritance; "virtual subclasses" registered with the ABC's register() method are not affected.

  Usage:

```
class C:
    __metaclass__ = ABCMeta
    @abstractmethod
    def my_abstract_method(self, ...):
        ...
```

  Note: Unlike Java abstract methods, these abstract methods may have an implementation. This implementation can be called via the super() mechanism from the class that overrides it. This could

be useful as an end-point for a super-call in a framework that uses cooperative multiple-inheritance.

abc.abstractproperty([fget[, fset[, fdel[, doc]]]])
A subclass of the built-in property(), indicating an abstract property.

Using this function requires that the class's metaclass is ABCMeta or is derived from it. A class that has a metaclass derived from ABCMeta cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract properties can be called using any of the normal 'super' call mechanisms.

Usage:

```
class C:
    __metaclass__ = ABCMeta
    @abstractproperty
    def my_abstract_property(self):
        ...
```

This defines a read-only property; you can also define a read-write abstract property using the 'long' form of property declaration:

```
class C:
    __metaclass__ = ABCMeta
    def getx(self): ...
    def setx(self, value): ...
    x = abstractproperty(getx, setx)
```

28.9 atexit — Exit handlers

New in version 2.0.

Source code: Lib/atexit.py

The atexit module defines a single function to register cleanup functions. Functions thus registered are automatically executed upon normal interpreter termination. atexit runs these functions in the reverse order in which they were registered; if you register A, B, and C, at interpreter termination time they will be run in the order C, B, A.

Note: The functions registered via this module are not called when the program is killed by a signal not handled by Python, when a Python fatal internal error is detected, or when os._exit() is called.

This is an alternate interface to the functionality provided by the sys.exitfunc() variable.

Note: This module is unlikely to work correctly when used with other code that sets sys.exitfunc. In particular, other core Python modules are free to use atexit without the programmer's knowledge. Authors who use sys.exitfunc should convert their code to use atexit instead. The simplest way to convert code that sets sys.exitfunc is to import atexit and register the function that had been bound to sys.exitfunc.

atexit.register(func[, *args[, **kwargs]])
Register func as a function to be executed at termination. Any optional arguments that are to be passed to func must be passed as arguments to register(). It is possible to register the same function and arguments more than once.

At normal program termination (for instance, if sys.exit() is called or the main module's execution completes), all functions registered are called in last in, first out order. The assumption is that lower level modules will normally be imported before higher level modules and thus must be cleaned up later.

If an exception is raised during execution of the exit handlers, a traceback is printed (unless SystemExit is raised) and the exception information is saved. After all exit handlers have had a chance to run the last exception to be raised is re-raised.

Changed in version 2.6: This function now returns func, which makes it possible to use it as a decorator.

See also:

Module readline Useful example of atexit to read and write readline history files.

28.9.1 atexit Example

The following simple example demonstrates how a module can initialize a counter from a file when it is imported and save the counter's updated value automatically when the program terminates without relying on the application making an explicit call into this module at termination.

```
try:
    _count = int(open("counter").read())
except IOError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    open("counter", "w").write("%d" % _count)

import atexit
atexit.register(savecounter)
```

Positional and keyword arguments may also be passed to register() to be passed along to the registered function when it is called:

```
def goodbye(name, adjective):
    print 'Goodbye, %s, it was %s to meet you.' % (name, adjective)

import atexit
atexit.register(goodbye, 'Donny', 'nice')

# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

Usage as a decorator:

```
import atexit

@atexit.register
def goodbye():
    print "You are now leaving the Python sector."
```

This only works with functions that can be called without arguments.

28.10 traceback — Print or retrieve a stack traceback

This module provides a standard interface to extract, format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. This is useful when you want to print stack traces under program control, such as in a "wrapper" around the interpreter.

The module uses traceback objects — this is the object type that is stored in the variables sys.exc_traceback (deprecated) and sys.last_traceback and returned as the third item from sys.exc_info().

The module defines the following functions:

traceback.print_tb(tb[, limit[, file]])
: Print up to limit stack trace entries from the traceback object tb. If limit is omitted or None, all entries are printed. If file is omitted or None, the output goes to sys.stderr; otherwise it should be an open file or file-like object to receive the output.

traceback.print_exception(etype, value, tb[, limit[, file]])
: Print exception information and up to limit stack trace entries from the traceback tb to file. This differs from print_tb() in the following ways: (1) if tb is not None, it prints a header Traceback (most recent call last):; (2) it prints the exception etype and value after the stack trace; (3) if etype is SyntaxError and value has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

traceback.print_exc([limit[, file]])
: This is a shorthand for print_exception(sys.exc_type, sys.exc_value, sys.exc_traceback, limit, file). (In fact, it uses sys.exc_info() to retrieve the same information in a thread-safe way instead of using the deprecated variables.)

traceback.format_exc([limit])
: This is like print_exc(limit) but returns a string instead of printing to a file.

  New in version 2.4.

traceback.print_last([limit[, file]])
: This is a shorthand for print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file). In general it will work only after an exception has reached an interactive prompt (see sys.last_type).

traceback.print_stack([f[, limit[, file]]])
: This function prints a stack trace from its invocation point. The optional f argument can be used to specify an alternate stack frame to start. The optional limit and file arguments have the same meaning as for print_exception().

traceback.extract_tb(tb[, limit])
: Return a list of up to limit "pre-processed" stack trace entries extracted from the traceback object tb. It is useful for alternate formatting of stack traces. If limit is omitted or None, all entries are extracted. A "pre-processed" stack trace entry is a 4-tuple (filename, line number, function name*, text) representing the information that is usually printed for a stack trace. The text is a string with leading and trailing whitespace stripped; if the source is not available it is None.

traceback.extract_stack([f[, limit]])
: Extract the raw traceback from the current stack frame. The return value has the same format as for extract_tb(). The optional f and limit arguments have the same meaning as for print_stack().

traceback.format_list(extracted_list)
: Given a list of tuples as returned by extract_tb() or extract_stack(), return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not None.

traceback.format_exception_only(etype, value)
: Format the exception part of a traceback. The arguments are the exception type, etype and value such as given by sys.last_type and sys.last_value. The return value is a list of strings, each ending in a newline. Normally, the list contains a single string; however, for SyntaxError exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. The message indicating which exception occurred is the always last string in the list.

traceback.format_exception(etype, value, tb[, limit])
: Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to print_exception(). The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does print_exception().

traceback.format_tb(tb[, limit])
: A shorthand for format_list(extract_tb(tb, limit)).

traceback.format_stack([f[, limit]])
: A shorthand for format_list(extract_stack(f, limit)).

traceback.tb_lineno(tb)
: This function returns the current line number set in the traceback object. This function was necessary because in versions of Python prior to 2.3 when the -O flag was passed to Python the tb.tb_lineno was not updated correctly. This function has no use in versions past 2.3.

28.10.1 Traceback Examples

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the code module.

```
import sys, traceback

def run_user_code(envdir):
    source = raw_input(">>> ")
    try:
        exec source in envdir
    except:
        print "Exception in user code:"
        print '-'*60
        traceback.print_exc(file=sys.stdout)
        print '-'*60

envdir = {}
while 1:
    run_user_code(envdir)
```

The following example demonstrates the different ways to print and format the exception and traceback:

```
import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]
```

(continues on next page)

(continued from previous page)

```
try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print "*** print_tb:"
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print "*** print_exception:"
    traceback.print_exception(exc_type, exc_value, exc_traceback,
                              limit=2, file=sys.stdout)
    print "*** print_exc:"
    traceback.print_exc()
    print "*** format_exc, first and last line:"
    formatted_lines = traceback.format_exc().splitlines()
    print formatted_lines[0]
    print formatted_lines[-1]
    print "*** format_exception:"
    print repr(traceback.format_exception(exc_type, exc_value,
                                          exc_traceback))
    print "*** extract_tb:"
    print repr(traceback.extract_tb(exc_traceback))
    print "*** format_tb:"
    print repr(traceback.format_tb(exc_traceback))
    print "*** tb_lineno:", exc_traceback.tb_lineno
```

The output for the example would look similar to this:

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[('<doctest...>', 10, '<module>', 'lumberjack()'),
 ('<doctest...>', 4, 'lumberjack', 'bright_side_of_death()'),
 ('<doctest...>', 7, 'bright_side_of_death', 'return tuple()[0]')]
```

(continues on next page)

(continued from previous page)

```
*** format_tb:
['  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n']
*** tb_lineno: 10
```

The following example shows the different ways to print and format the stack:

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print repr(traceback.extract_stack())
...     print repr(traceback.format_stack())
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print repr(traceback.extract_stack())')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print repr(traceback.format_stack())\n']
```

This last example demonstrates the final few formatting functions:

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(type(an_error), an_error)
['IndexError: tuple index out of range\n']
```

28.11 __future__ — Future statement definitions

Source code: Lib/__future__.py

__future__ is a real module, and serves three purposes:

- To avoid confusing existing tools that analyze import statements and expect to find the modules they're importing.
- To ensure that future statements run under releases prior to 2.1 at least yield runtime exceptions (the import of __future__ will fail, because there was no module of that name prior to 2.1).

- To document when incompatible changes were introduced, and when they will be — or were — made mandatory. This is a form of executable documentation, and can be inspected programmatically via importing __future__ and examining its contents.

Each statement in __future__.py is of the form:

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

where, normally, OptionalRelease is less than MandatoryRelease, and both are 5-tuples of the same form as sys.version_info:

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

OptionalRelease records the first release in which the feature was accepted.

In the case of a MandatoryRelease that has not yet occurred, MandatoryRelease predicts the release in which the feature will become part of the language.

Else MandatoryRelease records when the feature became part of the language; in releases at or after that, modules no longer need a future statement to use the feature in question, but may continue to use such imports.

MandatoryRelease may also be None, meaning that a planned feature got dropped.

Instances of class _Feature have two corresponding methods, getOptionalRelease() and getMandatoryRelease().

CompilerFlag is the (bitfield) flag that should be passed in the fourth argument to the built-in function compile() to enable the feature in dynamically compiled code. This flag is stored in the compiler_flag attribute on _Feature instances.

No feature description will ever be deleted from __future__. Since its introduction in Python 2.1 the following features have found their way into the language using this mechanism:

feature	optional in	mandatory in	effect
nested_scopes	2.1.0b1	2.2	PEP 227: Statically Nested Scopes
generators	2.2.0a1	2.3	PEP 255: Simple Generators
division	2.2.0a2	3.0	PEP 238: Changing the Division Operator
absolute_import	2.5.0a1	3.0	PEP 328: Imports: Multi-Line and Absolute/Relative
with_statement	2.5.0a1	2.6	PEP 343: The “with” Statement
print_function	2.6.0a2	3.0	PEP 3105: Make print a function
unicode_literals	2.6.0a2	3.0	PEP 3112: Bytes literals in Python 3000

See also:

future How the compiler treats future imports.

28.12 gc — Garbage Collector interface

This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disable the collector if you are sure your program does not create reference cycles. Automatic collection can be disabled by calling gc.disable(). To debug a leaking program call gc.set_debug(gc.DEBUG_LEAK). Notice that this includes gc.DEBUG_SAVEALL, causing garbage-collected objects to be saved in gc.garbage for inspection.

The gc module provides the following functions:

gc.enable()
: Enable automatic garbage collection.

gc.disable()
: Disable automatic garbage collection.

gc.isenabled()
: Returns true if automatic collection is enabled.

gc.collect([generation])
: With no arguments, run a full collection. The optional argument generation may be an integer specifying which generation to collect (from 0 to 2). A ValueError is raised if the generation number is invalid. The number of unreachable objects found is returned.

  Changed in version 2.5: The optional generation argument was added.

  Changed in version 2.6: The free lists maintained for a number of built-in types are cleared whenever a full collection or collection of the highest generation (2) is run. Not all items in some free lists may be freed due to the particular implementation, in particular int and float.

gc.set_debug(flags)
: Set the garbage collection debugging flags. Debugging information will be written to sys.stderr. See below for a list of debugging flags which can be combined using bit operations to control debugging.

gc.get_debug()
: Return the debugging flags currently set.

gc.get_objects()
: Returns a list of all objects tracked by the collector, excluding the list returned.

  New in version 2.2.

gc.set_threshold(threshold0[, threshold1[, threshold2]])
: Set the garbage collection thresholds (the collection frequency). Setting threshold0 to zero disables collection.

  The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation 0). If an object survives a collection it is moved into the next older generation. Since generation 2 is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds threshold0, collection starts. Initially only generation 0 is examined. If generation 0 has been examined more than threshold1 times since generation 1 has been examined, then generation 1 is examined as well. Similarly, threshold2 controls the number of collections of generation 1 before collecting generation 2.

gc.get_count()
: Return the current collection counts as a tuple of (count0, count1, count2).

New in version 2.5.

gc.get_threshold()
Return the current collection thresholds as a tuple of (threshold0, threshold1, threshold2).

gc.get_referrers(*objs)
Return the list of objects that directly refer to any of objs. This function will only locate those containers which support garbage collection; extension types which do refer to other objects but do not support garbage collection will not be found.

Note that objects which have already been dereferenced, but which live in cycles and have not yet been collected by the garbage collector can be listed among the resulting referrers. To get only currently live objects, call collect() before calling get_referrers().

Care must be taken when using objects returned by get_referrers() because some of them could still be under construction and hence in a temporarily invalid state. Avoid using get_referrers() for any purpose other than debugging.

New in version 2.2.

gc.get_referents(*objs)
Return a list of objects directly referred to by any of the arguments. The referents returned are those objects visited by the arguments' C-level tp_traverse methods (if any), and may not be all objects actually directly reachable. tp_traverse methods are supported only by objects that support garbage collection, and are only required to visit objects that may be involved in a cycle. So, for example, if an integer is directly reachable from an argument, that integer object may or may not appear in the result list.

New in version 2.3.

gc.is_tracked(obj)
Returns True if the object is currently tracked by the garbage collector, False otherwise. As a general rule, instances of atomic types aren't tracked and instances of non-atomic types (containers, user-defined objects...) are. However, some type-specific optimizations can be present in order to suppress the garbage collector footprint of simple instances (e.g. dicts containing only atomic keys and values):

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

New in version 2.7.

The following variable is provided for read-only access (you can mutate its value but should not rebind it):

gc.garbage
A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). By default, this list contains only objects with __del__() methods.[1] Objects that have __del__() methods and are part of a reference cycle cause the entire reference cycle to be uncollectable, including objects not necessarily in the cycle but reachable only from it. Python doesn't collect such cycles automatically because, in general, it isn't possible for Python to guess a safe order

[1] Prior to Python 2.2, the list contained all instance objects in unreachable cycles, not only those with __del__() methods.

in which to run the __del__() methods. If you know a safe order, you can force the issue by examining the garbage list, and explicitly breaking cycles due to your objects within the list. Note that these objects are kept alive even so by virtue of being in the garbage list, so they should be removed from garbage too. For example, after breaking cycles, do del gc.garbage[:] to empty the list. It's generally better to avoid the issue by not creating cycles containing objects with __del__() methods, and garbage can be examined in that case to verify that no such cycles are being created.

If DEBUG_SAVEALL is set, then all unreachable objects will be added to this list rather than freed.

The following constants are provided for use with set_debug():

gc.DEBUG_STATS
Print statistics during collection. This information can be useful when tuning the collection frequency.

gc.DEBUG_COLLECTABLE
Print information on collectable objects found.

gc.DEBUG_UNCOLLECTABLE
Print information of uncollectable objects found (objects which are not reachable but cannot be freed by the collector). These objects will be added to the garbage list.

gc.DEBUG_INSTANCES
When DEBUG_COLLECTABLE or DEBUG_UNCOLLECTABLE is set, print information about instance objects found.

gc.DEBUG_OBJECTS
When DEBUG_COLLECTABLE or DEBUG_UNCOLLECTABLE is set, print information about objects other than instance objects found.

gc.DEBUG_SAVEALL
When set, all unreachable objects found will be appended to garbage rather than being freed. This can be useful for debugging a leaking program.

gc.DEBUG_LEAK
The debugging flags necessary for the collector to print information about a leaking program (equal to DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_INSTANCES | DEBUG_OBJECTS | DEBUG_SAVEALL).

28.13 inspect — Inspect live objects

New in version 2.1.

Source code: Lib/inspect.py

The inspect module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module: type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.

28.13.1 Types and members

The getmembers() function retrieves the members of an object such as a class or module. The sixteen functions whose names begin with "is" are mainly provided as convenient choices for the second argument to

getmembers(). They also help you determine when you can expect to find the following special attributes:

Type	Attribute	Description
module	__doc__	documentation string
	__file__	filename (missing for built-in modules)
class	__doc__	documentation string
	__module__	name of module in which this class was defined
method	__doc__	documentation string
	__name__	name with which this method was defined
	im_class	class object that asked for this method
	im_func or __func__	function object containing implementation of method
	im_self or __self__	instance to which this method is bound, or None
function	__doc__	documentation string
	__name__	name with which this function was defined
	func_code	code object containing compiled function bytecode
	func_defaults	tuple of any default values for arguments
	func_doc	(same as __doc__)
	func_globals	global namespace in which this function was defined
	func_name	(same as __name__)
generator	__iter__	defined to support iteration over container
	close	raises new GeneratorExit exception inside the generator to terminate the iteration
	gi_code	code object
	gi_frame	frame object or possibly None once the generator has been exhausted
	gi_running	set to 1 when generator is executing, 0 otherwise
	next	return the next item from the container
	send	resumes the generator and "sends" a value that becomes the result of the current yi
	throw	used to raise an exception inside the generator
traceback	tb_frame	frame object at this level
	tb_lasti	index of last attempted instruction in bytecode
	tb_lineno	current line number in Python source code
	tb_next	next inner traceback object (called by this level)
frame	f_back	next outer frame object (this frame's caller)
	f_builtins	builtins namespace seen by this frame
	f_code	code object being executed in this frame
	f_exc_traceback	traceback if raised in this frame, or None
	f_exc_type	exception type if raised in this frame, or None
	f_exc_value	exception value if raised in this frame, or None
	f_globals	global namespace seen by this frame
	f_lasti	index of last attempted instruction in bytecode
	f_lineno	current line number in Python source code
	f_locals	local namespace seen by this frame
	f_restricted	0 or 1 if frame is in restricted execution mode
	f_trace	tracing function for this frame, or None
code	co_argcount	number of arguments (not including * or ** args)
	co_code	string of raw compiled bytecode
	co_consts	tuple of constants used in the bytecode
	co_filename	name of file in which this code object was created
	co_firstlineno	number of first line in Python source code
	co_flags	bitmap: 1=optimized | 2=newlocals | 4=*arg | 8=**arg
	co_lnotab	encoded mapping of line numbers to bytecode indices
	co_name	name with which this code object was defined

C

Table 1 – continued from previous page

Type	Attribute	Description
	co_names	tuple of names of local variables
	co_nlocals	number of local variables
	co_stacksize	virtual machine stack space required
	co_varnames	tuple of names of arguments and local variables
builtin	__doc__	documentation string
	__name__	original name of this function or method
	__self__	instance to which a method is bound, or None

Note:

(1) Changed in version 2.2: im_class used to refer to the class that defined the method.

inspect.getmembers(object[, predicate])
Return all the members of an object in a list of (name, value) pairs sorted by name. If the optional predicate argument is supplied, only members for which the predicate returns a true value are included.

Note: getmembers() does not return metaclass attributes when the argument is a class (this behavior is inherited from the dir() function).

inspect.getmoduleinfo(path)
Return a tuple of values that describe how Python will interpret the file identified by path if it is a module, or None if it would not be identified as a module. The return tuple is (name, suffix, mode, module_type), where name is the name of the module without the name of any enclosing package, suffix is the trailing part of the file name (which may not be a dot-delimited extension), mode is the open() mode that would be used (`'r'` or `'rb'`), and module_type is an integer giving the type of the module. module_type will have a value which can be compared to the constants defined in the imp module; see the documentation for that module for more information on module types.

Changed in version 2.6: Returns a named tuple ModuleInfo(name, suffix, mode, module_type).

inspect.getmodulename(path)
Return the name of the module named by the file path, without including the names of enclosing packages. This uses the same algorithm as the interpreter uses when searching for modules. If the name cannot be matched according to the interpreter's rules, None is returned.

inspect.ismodule(object)
Return true if the object is a module.

inspect.isclass(object)
Return true if the object is a class, whether built-in or created in Python code.

inspect.ismethod(object)
Return true if the object is a bound or unbound method written in Python.

inspect.isfunction(object)
Return true if the object is a Python function, which includes functions created by a lambda expression.

inspect.isgeneratorfunction(object)
Return true if the object is a Python generator function.

New in version 2.6.

inspect.isgenerator(object)
Return true if the object is a generator.

New in version 2.6.

inspect.istraceback(object)
Return true if the object is a traceback.

inspect.isframe(object)
Return true if the object is a frame.

inspect.iscode(object)
Return true if the object is a code.

inspect.isbuiltin(object)
Return true if the object is a built-in function or a bound built-in method.

inspect.isroutine(object)
Return true if the object is a user-defined or built-in function or method.

inspect.isabstract(object)
Return true if the object is an abstract base class.

New in version 2.6.

inspect.ismethoddescriptor(object)
Return true if the object is a method descriptor, but not if ismethod(), isclass(), isfunction() or isbuiltin() are true.

This is new as of Python 2.2, and, for example, is true of int.__add__. An object passing this test has a __get__() method but not a __set__() method, but beyond that the set of attributes varies. A __name__ attribute is usually sensible, and __doc__ often is.

Methods implemented via descriptors that also pass one of the other tests return false from the ismethoddescriptor() test, simply because the other tests promise more – you can, e.g., count on having the im_func attribute (etc) when an object passes ismethod().

inspect.isdatadescriptor(object)
Return true if the object is a data descriptor.

Data descriptors have both a __get__ and a __set__ method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have __name__ and __doc__ attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

New in version 2.3.

inspect.isgetsetdescriptor(object)
Return true if the object is a getset descriptor.

CPython implementation detail: getsets are attributes defined in extension modules via PyGetSetDef structures. For Python implementations without such types, this method will always return False.

New in version 2.5.

inspect.ismemberdescriptor(object)
Return true if the object is a member descriptor.

CPython implementation detail: Member descriptors are attributes defined in extension modules via PyMemberDef structures. For Python implementations without such types, this method will always return False.

New in version 2.5.

28.13.2 Retrieving source code

inspect.getdoc(object)
Get the documentation string for an object, cleaned up with cleandoc().

inspect.getcomments(object)
Return in a single string any lines of comments immediately preceding the object's source code (for a class, function, or method), or at the top of the Python source file (if the object is a module).

inspect.getfile(object)
Return the name of the (text or binary) file in which an object was defined. This will fail with a TypeError if the object is a built-in module, class, or function.

inspect.getmodule(object)
Try to guess which module an object was defined in.

inspect.getsourcefile(object)
Return the name of the Python source file in which an object was defined. This will fail with a TypeError if the object is a built-in module, class, or function.

inspect.getsourcelines(object)
Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An IOError is raised if the source code cannot be retrieved.

inspect.getsource(object)
Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An IOError is raised if the source code cannot be retrieved.

inspect.cleandoc(doc)
Clean up indentation from docstrings that are indented to line up with blocks of code.

All leading whitespace is removed from the first line. Any leading whitespace that can be uniformly removed from the second line onwards is removed. Empty lines at the beginning and end are subsequently removed. Also, all tabs are expanded to spaces.

New in version 2.6.

28.13.3 Classes and functions

inspect.getclasstree(classes[, unique])
Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the unique argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

inspect.getargspec(func)
Get the names and default values of a Python function's arguments. A tuple of four things is returned: (args, varargs, keywords, defaults). args is a list of the argument names (it may contain nested lists). varargs and keywords are the names of the * and ** arguments or None. defaults is a tuple of default argument values or None if there are no default arguments; if this tuple has n elements, they correspond to the last n elements listed in args.

Changed in version 2.6: Returns a named tuple ArgSpec(args, varargs, keywords, defaults).

inspect.getargvalues(frame)
Get information about arguments passed into a particular frame. A tuple of four things is returned: (args, varargs, keywords, locals). args is a list of the argument names (it may contain nested lists). varargs and keywords are the names of the * and ** arguments or None. locals is the locals dictionary of the given frame.

Changed in version 2.6: Returns a named tuple ArgInfo(args, varargs, keywords, locals).

inspect.formatargspec(args[, varargs, varkw, defaults, formatarg, formatvarargs, formatvarkw, formatvalue, join])
Format a pretty argument spec from the four values returned by getargspec(). The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

inspect.formatargvalues(args[, varargs, varkw, locals, formatarg, formatvarargs, formatvarkw, formatvalue, join])
Format a pretty argument spec from the four values returned by getargvalues(). The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

inspect.getmro(cls)
Return a tuple of class cls's base classes, including cls, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on cls's type. Unless a very peculiar user-defined metatype is in use, cls will be the first element of the tuple.

inspect.getcallargs(func[, *args][, **kwds])
Bind the args and kwds to the argument names of the Python function or method func, as if it was called with them. For bound methods, bind also the first argument (typically named self) to the associated instance. A dict is returned, mapping the argument names (including the names of the * and ** arguments, if any) to their values from args and kwds. In case of invoking func incorrectly, i.e. whenever func(*args, **kwds) would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3)
{'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
>>> getcallargs(f, a=2, x=4)
{'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() takes at least 1 argument (0 given)
```

New in version 2.7.

28.13.4 The interpreter stack

When the following functions return "frame records," each record is a tuple of six items: the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list.

Note: Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the

lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python's optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a finally clause. This is also important if the cycle detector was disabled when Python was compiled or using gc.disable(). For example:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

The optional context argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

inspect.getframeinfo(frame[, context])
: Get information about a frame or traceback object. A 5-tuple is returned, the last five elements of the frame's frame record.

  Changed in version 2.6: Returns a named tuple Traceback(filename, lineno, function, code_context, index).

inspect.getouterframes(frame[, context])
: Get a list of frame records for a frame and all outer frames. These frames represent the calls that lead to the creation of frame. The first entry in the returned list represents frame; the last entry represents the outermost call on frame's stack.

inspect.getinnerframes(traceback[, context])
: Get a list of frame records for a traceback's frame and all inner frames. These frames represent calls made as a consequence of frame. The first entry in the list represents traceback; the last entry represents where the exception was raised.

inspect.currentframe()
: Return the frame object for the caller's stack frame.

  CPython implementation detail: This function relies on Python stack frame support in the interpreter, which isn't guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns None.

inspect.stack([context])
: Return a list of frame records for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

inspect.trace([context])
: Return a list of frame records for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

28.14 site — Site-specific configuration hook

Source code: Lib/site.py

This module is automatically imported during initialization. The automatic import can be suppressed using the interpreter's -S option.

Importing this module will append site-specific paths to the module search path and add a few builtins.

It starts by constructing up to four directories from a head and a tail part. For the head part, it uses sys.prefix and sys.exec_prefix; empty heads are skipped. For the tail part, it uses the empty string and then lib/site-packages (on Windows) or lib/pythonX.Y/site-packages and then lib/site-python (on Unix and Macintosh). For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds it to sys.path and also inspects the newly added path for configuration files.

A path configuration file is a file whose name has the form name.pth and exists in one of the four directories mentioned above; its contents are additional items (one per line) to be added to sys.path. Non-existing items are never added to sys.path, and no check is made that the item refers to a directory rather than a file. No item is added to sys.path more than once. Blank lines and lines beginning with # are skipped. Lines starting with import (followed by space or tab) are executed.

Changed in version 2.6: A space or tab is now required after the import keyword.

For example, suppose sys.prefix and sys.exec_prefix are set to /usr/local. The Python X.Y library is then installed in /usr/local/lib/pythonX.Y. Suppose this has a subdirectory /usr/local/lib/pythonX.Y/site-packages with three subsubdirectories, foo, bar and spam, and two path configuration files, foo.pth and bar.pth. Assume foo.pth contains the following:

```
# foo package configuration

foo
bar
bletch
```

and bar.pth contains:

```
# bar package configuration

bar
```

Then the following version-specific directories are added to sys.path, in this order:

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

Note that bletch is omitted because it doesn't exist; the bar directory precedes the foo directory because bar.pth comes alphabetically before foo.pth; and spam is omitted because it is not mentioned in either path configuration file.

After these path manipulations, an attempt is made to import a module named sitecustomize, which can perform arbitrary site-specific customizations. It is typically created by a system administrator in the site-packages directory. If this import fails with an ImportError exception, it is silently ignored. If Python is started without output streams available, as with pythonw.exe on Windows (which is used by default to start IDLE), attempted output from sitecustomize is ignored. Any exception other than ImportError causes a silent and perhaps mysterious failure of the process.

After this, an attempt is made to import a module named usercustomize, which can perform arbitrary user-specific customizations, if ENABLE_USER_SITE is true. This file is intended to be created in the user site-packages directory (see below), which is part of sys.path unless disabled by -s. An ImportError will be silently ignored.

Note that for some non-Unix systems, sys.prefix and sys.exec_prefix are empty, and the path manipulations are skipped; however the import of sitecustomize and usercustomize is still attempted.

site.PREFIXES
: A list of prefixes for site-packages directories.

  New in version 2.6.

site.ENABLE_USER_SITE
: Flag showing the status of the user site-packages directory. True means that it is enabled and was added to sys.path. False means that it was disabled by user request (with -s or PYTHONNOUSERSITE). None means it was disabled for security reasons (mismatch between user or group id and effective id) or by an administrator.

  New in version 2.6.

site.USER_SITE
: Path to the user site-packages for the running Python. Can be None if getusersitepackages() hasn't been called yet. Default value is ~/.local/lib/pythonX.Y/site-packages for UNIX and non-framework Mac OS X builds, ~/Library/Python/X.Y/lib/python/site-packages for Mac framework builds, and %APPDATA%\Python\PythonXY\site-packages on Windows. This directory is a site directory, which means that .pth files in it will be processed.

  New in version 2.6.

site.USER_BASE
: Path to the base directory for the user site-packages. Can be None if getuserbase() hasn't been called yet. Default value is ~/.local for UNIX and Mac OS X non-framework builds, ~/Library/Python/X.Y for Mac framework builds, and %APPDATA%\Python for Windows. This value is used by Distutils to compute the installation directories for scripts, data files, Python modules, etc. for the user installation scheme. See also PYTHONUSERBASE.

  New in version 2.6.

site.addsitedir(sitedir, known_paths=None)
: Add a directory to sys.path and process its .pth files. Typically used in sitecustomize or usercustomize (see above).

site.getsitepackages()
: Return a list containing all global site-packages directories (and possibly site-python).

  New in version 2.7.

site.getuserbase()
: Return the path of the user base directory, USER_BASE. If it is not initialized yet, this function will also set it, respecting PYTHONUSERBASE.

  New in version 2.7.

site.getusersitepackages()
: Return the path of the user-specific site-packages directory, USER_SITE. If it is not initialized yet, this function will also set it, respecting PYTHONNOUSERSITE and USER_BASE.

  New in version 2.7.

The site module also provides a way to get the user directories from the command line:

```
$ python -m site --user-site
/home/user/.local/lib/python2.7/site-packages
```

If it is called without arguments, it will print the contents of sys.path on the standard output, followed by the value of USER_BASE and whether the directory exists, then the same thing for USER_SITE, and finally the value of ENABLE_USER_SITE.

--user-base
: Print the path to the user base directory.

--user-site
: Print the path to the user site-packages directory.

If both options are given, user base and user site will be printed (always in this order), separated by os.pathsep.

If any option is given, the script will exit with one of these values: 0 if the user site-packages directory is enabled, 1 if it was disabled by the user, 2 if it is disabled for security reasons or by an administrator, and a value greater than 2 if there is an error.

See also:

PEP 370 – Per user site-packages directory

28.15 user — User-specific configuration hook

Deprecated since version 2.6: The user module has been removed in Python 3.

As a policy, Python doesn't run user-specified code on startup of Python programs. (Only interactive sessions execute the script specified in the PYTHONSTARTUP environment variable if it exists).

However, some programs or sites may find it convenient to allow users to have a standard customization file, which gets run when a program requests it. This module implements such a mechanism. A program that wishes to use the mechanism must execute the statement

```
import user
```

The user module looks for a file .pythonrc.py in the user's home directory and if it can be opened, executes it (using execfile()) in its own (the module user's) global namespace. Errors during this phase are not caught; that's up to the program that imports the user module, if it wishes. The home directory is assumed to be named by the HOME environment variable; if this is not set, the current directory is used.

The user's .pythonrc.py could conceivably test for sys.version if it wishes to do different things depending on the Python version.

A warning to users: be very conservative in what you place in your .pythonrc.py file. Since you don't know which programs will use it, changing the behavior of standard modules or functions is generally not a good idea.

A suggestion for programmers who wish to use this mechanism: a simple way to let users specify options for your package is to have them define variables in their .pythonrc.py file that you test in your module. For example, a module spam that has a verbosity level can look for a variable user.spam_verbose, as follows:

```
import user

verbose = bool(getattr(user, "spam_verbose", 0))
```

(The three-argument form of getattr() is used in case the user has not defined spam_verbose in their .pythonrc.py file.)

Programs with extensive customization needs are better off reading a program-specific customization file.

Programs with security or privacy concerns should not import this module; a user can easily break into a program by placing arbitrary code in the .pythonrc.py file.

Modules for general use should not import this module; it may interfere with the operation of the importing program.

See also:

Module site Site-wide customization mechanism.

28.16 fpectl — Floating point exception control

Note: The fpectl module is not built by default, and its usage is discouraged and may be dangerous except in the hands of experts. See also the section Limitations and other considerations on limitations for more details.

Most computers carry out floating point operations in conformance with the so-called IEEE-754 standard. On any real computer, some floating point operations produce results that cannot be expressed as a normal floating point value. For example, try

```
>>> import math
>>> math.exp(1000)
inf
>>> math.exp(1000) / math.exp(1000)
nan
```

(The example above will work on many platforms. DEC Alpha may be one exception.) "Inf" is a special, non-numeric value in IEEE-754 that stands for "infinity", and "nan" means "not a number." Note that, other than the non-numeric results, nothing special happened when you asked Python to carry out those calculations. That is in fact the default behaviour prescribed in the IEEE-754 standard, and if it works for you, stop reading now.

In some circumstances, it would be better to raise an exception and stop processing at the point where the faulty operation was attempted. The fpectl module is for use in that situation. It provides control over floating point units from several hardware manufacturers, allowing the user to turn on the generation of SIGFPE whenever any of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid Operation occurs. In tandem with a pair of wrapper macros that are inserted into the C code comprising your python system, SIGFPE is trapped and converted into the Python FloatingPointError exception.

The fpectl module defines the following functions and may raise the given exception:

fpectl.turnon_sigfpe()
: Turn on the generation of SIGFPE, and set up an appropriate signal handler.

fpectl.turnoff_sigfpe()
: Reset default handling of floating point exceptions.

exception fpectl.FloatingPointError
: After turnon_sigfpe() has been executed, a floating point operation that raises one of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid operation will in turn raise this standard Python exception.

28.16.1 Example

The following example demonstrates how to start up and test operation of the fpectl module.

```
>>> import fpectl
>>> import fpetest
>>> fpectl.turnon_sigfpe()
```

(continues on next page)

(continued from previous page)

```
>>> fpetest.test()
overflow        PASS
FloatingPointError: Overflow

div by 0        PASS
FloatingPointError: Division by zero
  [ more output from test elided ]
>>> import math
>>> math.exp(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: in math_1
```

28.16.2 Limitations and other considerations

Setting up a given processor to trap IEEE-754 floating point errors currently requires custom code on a per-architecture basis. You may have to modify fpectl to control your particular hardware.

Conversion of an IEEE-754 exception to a Python exception requires that the wrapper macros PyFPE_START_PROTECT and PyFPE_END_PROTECT be inserted into your code in an appropriate fashion. Python itself has been modified to support the fpectl module, but many other codes of interest to numerical analysts have not.

The fpectl module is not thread-safe.

See also:

Some files in the source distribution may be interesting in learning more about how this module operates. The include file Include/pyfpe.h discusses the implementation of this module at some length. Modules/fpetestmodule.c gives several examples of use. Many additional examples can be found in Objects/floatobject.c.

CHAPTER

TWENTYNINE

CUSTOM PYTHON INTERPRETERS

The modules described in this chapter allow writing interfaces similar to Python's interactive interpreter. If you want a Python interpreter that supports some special feature in addition to the Python language, you should look at the code module. (The codeop module is lower-level, used to support compiling a possibly-incomplete chunk of Python code.)

The full list of modules described in this chapter is:

29.1 code — Interpreter base classes

The code module provides facilities to implement read-eval-print loops in Python. Two classes and convenience functions are included which can be used to build applications which provide an interactive interpreter prompt.

class code.InteractiveInterpreter([locals])
: This class deals with parsing and interpreter state (the user's namespace); it does not deal with input buffering or prompting or input file naming (the filename is always passed in explicitly). The optional locals argument specifies the dictionary in which code will be executed; it defaults to a newly created dictionary with key `'__name__'` set to `'__console__'` and key `'__doc__'` set to None.

class code.InteractiveConsole([locals[, filename]])
: Closely emulate the behavior of the interactive Python interpreter. This class builds on InteractiveInterpreter and adds prompting using the familiar sys.ps1 and sys.ps2, and input buffering.

code.interact([banner[, readfunc[, local]]])
: Convenience function to run a read-eval-print loop. This creates a new instance of InteractiveConsole and sets readfunc to be used as the InteractiveConsole.raw_input() method, if provided. If local is provided, it is passed to the InteractiveConsole constructor for use as the default namespace for the interpreter loop. The interact() method of the instance is then run with banner passed as the banner to use, if provided. The console object is discarded after use.

code.compile_command(source[, filename[, symbol]])
: This function is useful for programs that want to emulate Python's interpreter main loop (a.k.a. the read-eval-print loop). The tricky part is to determine when the user has entered an incomplete command that can be completed by entering more text (as opposed to a complete command or a syntax error). This function almost always makes the same decision as the real interpreter main loop.

  source is the source string; filename is the optional filename from which source was read, defaulting to `'<input>'`; and symbol is the optional grammar start symbol, which should be either `'single'` (the default) or `'eval'`.

  Returns a code object (the same as compile(source, filename, symbol)) if the command is complete and valid; None if the command is incomplete; raises SyntaxError if the command is complete and contains

a syntax error, or raises OverflowError or ValueError if the command contains an invalid literal.

29.1.1 Interactive Interpreter Objects

InteractiveInterpreter.runsource(source[, filename[, symbol]])
Compile and run some source in the interpreter. Arguments are the same as for compile_command(); the default for filename is '<input>', and for symbol is 'single'. One several things can happen:

- The input is incorrect; compile_command() raised an exception (SyntaxError or OverflowError). A syntax traceback will be printed by calling the showsyntaxerror() method. runsource() returns False.
- The input is incomplete, and more input is required; compile_command() returned None. runsource() returns True.
- The input is complete; compile_command() returned a code object. The code is executed by calling the runcode() (which also handles run-time exceptions, except for SystemExit). runsource() returns False.

The return value can be used to decide whether to use sys.ps1 or sys.ps2 to prompt the next line.

InteractiveInterpreter.runcode(code)
Execute a code object. When an exception occurs, showtraceback() is called to display a traceback. All exceptions are caught except SystemExit, which is allowed to propagate.

A note about KeyboardInterrupt: this exception may occur elsewhere in this code, and may not always be caught. The caller should be prepared to deal with it.

InteractiveInterpreter.showsyntaxerror([filename])
Display the syntax error that just occurred. This does not display a stack trace because there isn't one for syntax errors. If filename is given, it is stuffed into the exception instead of the default filename provided by Python's parser, because it always uses '<string>' when reading from a string. The output is written by the write() method.

InteractiveInterpreter.showtraceback()
Display the exception that just occurred. We remove the first stack item because it is within the interpreter object implementation. The output is written by the write() method.

InteractiveInterpreter.write(data)
Write a string to the standard error stream (sys.stderr). Derived classes should override this to provide the appropriate output handling as needed.

29.1.2 Interactive Console Objects

The InteractiveConsole class is a subclass of InteractiveInterpreter, and so offers all the methods of the interpreter objects as well as the following additions.

InteractiveConsole.interact([banner])
Closely emulate the interactive Python console. The optional banner argument specify the banner to print before the first interaction; by default it prints a banner similar to the one printed by the standard Python interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter – since it's so close!).

InteractiveConsole.push(line)
Push a line of source text to the interpreter. The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter's runsource() method is called with the concatenated contents of the buffer as source. If this indicates that the command was

executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is True if more input is required, False if the line was dealt with in some way (this is the same as runsource()).

InteractiveConsole.resetbuffer()
Remove any unhandled source text from the input buffer.

InteractiveConsole.raw_input([prompt])
Write a prompt and read a line. The returned line does not include the trailing newline. When the user enters the EOF key sequence, EOFError is raised. The base implementation uses the built-in function raw_input(); a subclass may replace this with a different implementation.

29.2 codeop — Compile Python code

The codeop module provides utilities upon which the Python read-eval-print loop can be emulated, as is done in the code module. As a result, you probably don't want to use the module directly; if you want to include such a loop in your program you probably want to use the code module instead.

There are two parts to this job:

1. Being able to tell if a line of input completes a Python statement: in short, telling whether to print '>>>' or '...' next.
2. Remembering which future statements the user has entered, so subsequent input can be compiled with these in effect.

The codeop module provides a way of doing each of these things, and a way of doing them both.

To do just the former:

codeop.compile_command(source[, filename[, symbol]])
Tries to compile source, which should be a string of Python code and return a code object if source is valid Python code. In that case, the filename attribute of the code object will be filename, which defaults to '<input>'. Returns None if source is not valid Python code, but is a prefix of valid Python code.

If there is a problem with source, an exception will be raised. SyntaxError is raised if there is invalid Python syntax, and OverflowError or ValueError if there is an invalid literal.

The symbol argument determines whether source is compiled as a statement ('single', the default) or as an expression ('eval'). Any other value will cause ValueError to be raised.

Note: It is possible (but not likely) that the parser stops parsing with a successful outcome before reaching the end of the source; in this case, trailing symbols may be ignored instead of causing an error. For example, a backslash followed by two newlines may be followed by arbitrary garbage. This will be fixed once the API for the parser is better.

---

class codeop.Compile
Instances of this class have __call__() methods identical in signature to the built-in function compile(), but with the difference that if the instance compiles program text containing a __future__ statement, the instance 'remembers' and compiles all subsequent program texts with the statement in force.

class codeop.CommandCompiler
Instances of this class have __call__() methods identical in signature to compile_command(); the

difference is that if the instance compiles program text containing a __future__ statement, the instance 'remembers' and compiles all subsequent program texts with the statement in force.

A note on version compatibility: the Compile and CommandCompiler are new in Python 2.2. If you want to enable the future-tracking features of 2.2 but also retain compatibility with 2.1 and earlier versions of Python you can either write

```
try:
    from codeop import CommandCompiler
    compile_command = CommandCompiler()
    del CommandCompiler
except ImportError:
    from codeop import compile_command
```

which is a low-impact change, but introduces possibly unwanted global state into your program, or you can write:

```
try:
    from codeop import CommandCompiler
except ImportError:
    def CommandCompiler():
        from codeop import compile_command
        return compile_command
```

and then call CommandCompiler every time you need a fresh compiler object.

CHAPTER

THIRTY

RESTRICTED EXECUTION

Warning: In Python 2.3 these modules have been disabled due to various known and not readily fixable security holes. The modules are still documented here to help in reading old code that uses the rexec and Bastion modules.

Restricted execution is the basic framework in Python that allows for the segregation of trusted and untrusted code. The framework is based on the notion that trusted Python code (a supervisor) can create a "padded cell' (or environment) with limited permissions, and run the untrusted code within this cell. The untrusted code cannot break out of its cell, and can only interact with sensitive system resources through interfaces defined and managed by the trusted code. The term "restricted execution" is favored over "safe-Python" since true safety is hard to define, and is determined by the way the restricted environment is created. Note that the restricted environments can be nested, with inner cells creating subcells of lesser, but never greater, privilege.

An interesting aspect of Python's restricted execution model is that the interfaces presented to untrusted code usually have the same names as those presented to trusted code. Therefore no special interfaces need to be learned to write code designed to run in a restricted environment. And because the exact nature of the padded cell is determined by the supervisor, different restrictions can be imposed, depending on the application. For example, it might be deemed "safe" for untrusted code to read any file within a specified directory, but never to write a file. In this case, the supervisor may redefine the built-in open() function so that it raises an exception whenever the mode parameter is `'w'`. It might also perform a chroot()-like operation on the filename parameter, such that root is always relative to some safe "sandbox" area of the filesystem. In this case, the untrusted code would still see a built-in open() function in its environment, with the same calling interface. The semantics would be identical too, with IOErrors being raised when the supervisor determined that an unallowable parameter is being used.

The Python run-time determines whether a particular code block is executing in restricted execution mode based on the identity of the __builtins__ object in its global variables: if this is (the dictionary of) the standard __builtin__ module, the code is deemed to be unrestricted, else it is deemed to be restricted.

Python code executing in restricted mode faces a number of limitations that are designed to prevent it from escaping from the padded cell. For instance, the function object attribute func_globals and the class and instance object attribute __dict__ are unavailable.

Two modules provide the framework for setting up restricted execution environments:

30.1 rexec — Restricted execution framework

Deprecated since version 2.6: The rexec module has been removed in Python 3.

Changed in version 2.3: Disabled module.

Warning: The documentation has been left in place to help in reading old code that uses the module.

This module contains the RExec class, which supports r_eval(), r_execfile(), r_exec(), and r_import() methods, which are restricted versions of the standard Python functions eval(), execfile() and the exec and import statements. Code executed in this restricted environment will only have access to modules and functions that are deemed safe; you can subclass RExec to add or remove capabilities as desired.

Warning: While the rexec module is designed to perform as described below, it does have a few known vulnerabilities which could be exploited by carefully written code. Thus it should not be relied upon in situations requiring "production ready" security. In such situations, execution via sub-processes or very careful "cleansing" of both code and data to be processed may be necessary. Alternatively, help in patching known rexec vulnerabilities would be welcomed.

Note: The RExec class can prevent code from performing unsafe operations like reading or writing disk files, or using TCP/IP sockets. However, it does not protect against code using extremely large amounts of memory or processor time.

class rexec.RExec([hooks[, verbose]])
Returns an instance of the RExec class.

hooks is an instance of the RHooks class or a subclass of it. If it is omitted or None, the default RHooks class is instantiated. Whenever the rexec module searches for a module (even a built-in one) or reads a module's code, it doesn't actually go out to the file system itself. Rather, it calls methods of an RHooks instance that was passed to or created by its constructor. (Actually, the RExec object doesn't make these calls — they are made by a module loader object that's part of the RExec object. This allows another level of flexibility, which can be useful when changing the mechanics of import within the restricted environment.)

By providing an alternate RHooks object, we can control the file system accesses made to import a module, without changing the actual algorithm that controls the order in which those accesses are made. For instance, we could substitute an RHooks object that passes all filesystem requests to a file server elsewhere, via some RPC mechanism such as ILU. Grail's applet loader uses this to support importing applets from a URL for a directory.

If verbose is true, additional debugging output may be sent to standard output.

It is important to be aware that code running in a restricted environment can still call the sys.exit() function. To disallow restricted code from exiting the interpreter, always protect calls that cause restricted code to run with a try/except statement that catches the SystemExit exception. Removing the sys.exit() function from the restricted environment is not sufficient — the restricted code could still use raise SystemExit. Removing SystemExit is not a reasonable option; some library code makes use of this and would break were it not available.

See also:

Grail Home Page Grail is a Web browser written entirely in Python. It uses the rexec module as a foundation for supporting Python applets, and can be used as an example usage of this module.

30.1.1 RExec Objects

RExec instances support the following methods:

RExec.r_eval(code)
code must either be a string containing a Python expression, or a compiled code object, which will be evaluated in the restricted environment's __main__ module. The value of the expression or code object will be returned.

RExec.r_exec(code)
code must either be a string containing one or more lines of Python code, or a compiled code object, which will be executed in the restricted environment's __main__ module.

RExec.r_execfile(filename)
Execute the Python code contained in the file filename in the restricted environment's __main__ module.

Methods whose names begin with s_ are similar to the functions beginning with r_, but the code will be granted access to restricted versions of the standard I/O streams sys.stdin, sys.stderr, and sys.stdout.

RExec.s_eval(code)
code must be a string containing a Python expression, which will be evaluated in the restricted environment.

RExec.s_exec(code)
code must be a string containing one or more lines of Python code, which will be executed in the restricted environment.

RExec.s_execfile(code)
Execute the Python code contained in the file filename in the restricted environment.

RExec objects must also support various methods which will be implicitly called by code executing in the restricted environment. Overriding these methods in a subclass is used to change the policies enforced by a restricted environment.

RExec.r_import(modulename[, globals[, locals[, fromlist]]])
Import the module modulename, raising an ImportError exception if the module is considered unsafe.

RExec.r_open(filename[, mode[, bufsize]])
Method called when open() is called in the restricted environment. The arguments are identical to those of open(), and a file object (or a class instance compatible with file objects) should be returned. RExec's default behaviour is allow opening any file for reading, but forbidding any attempt to write a file. See the example below for an implementation of a less restrictive r_open().

RExec.r_reload(module)
Reload the module object module, re-parsing and re-initializing it.

RExec.r_unload(module)
Unload the module object module (remove it from the restricted environment's sys.modules dictionary).

And their equivalents with access to restricted standard I/O streams:

RExec.s_import(modulename[, globals[, locals[, fromlist]]])
Import the module modulename, raising an ImportError exception if the module is considered unsafe.

RExec.s_reload(module)
Reload the module object module, re-parsing and re-initializing it.

RExec.s_unload(module)
Unload the module object module.

30.1.2 Defining restricted environments

The RExec class has the following class attributes, which are used by the __init__() method. Changing them on an existing instance won't have any effect; instead, create a subclass of RExec and assign them new values in the class definition. Instances of the new class will then use those new values. All these attributes are tuples of strings.

RExec.nok_builtin_names
: Contains the names of built-in functions which will not be available to programs running in the restricted environment. The value for RExec is ('open', 'reload', '__import__'). (This gives the exceptions, because by far the majority of built-in functions are harmless. A subclass that wants to override this variable should probably start with the value from the base class and concatenate additional forbidden functions — when new dangerous built-in functions are added to Python, they will also be added to this module.)

RExec.ok_builtin_modules
: Contains the names of built-in modules which can be safely imported. The value for RExec is ('audioop', 'array', 'binascii', 'cmath', 'errno', 'imageop', 'marshal', 'math', 'md5', 'operator', 'parser', 'regex', 'select', 'sha', '_sre', 'strop', 'struct', 'time'). A similar remark about overriding this variable applies — use the value from the base class as a starting point.

RExec.ok_path
: Contains the directories which will be searched when an import is performed in the restricted environment. The value for RExec is the same as sys.path (at the time the module is loaded) for unrestricted code.

RExec.ok_posix_names
: Contains the names of the functions in the os module which will be available to programs running in the restricted environment. The value for RExec is ('error', 'fstat', 'listdir', 'lstat', 'readlink', 'stat', 'times', 'uname', 'getpid', 'getppid', 'getcwd', 'getuid', 'getgid', 'geteuid', 'getegid').

RExec.ok_sys_names
: Contains the names of the functions and variables in the sys module which will be available to programs running in the restricted environment. The value for RExec is ('ps1', 'ps2', 'copyright', 'version', 'platform', 'exit', 'maxint').

RExec.ok_file_types
: Contains the file types from which modules are allowed to be loaded. Each file type is an integer constant defined in the imp module. The meaningful values are PY_SOURCE, PY_COMPILED, and C_EXTENSION. The value for RExec is (C_EXTENSION, PY_SOURCE). Adding PY_COMPILED in subclasses is not recommended; an attacker could exit the restricted execution mode by putting a forged byte-compiled file (.pyc) anywhere in your file system, for example by writing it to /tmp or uploading it to the /incoming directory of your public FTP server.

30.1.3 An example

Let us say that we want a slightly more relaxed policy than the standard RExec class. For example, if we're willing to allow files in /tmp to be written, we can subclass the RExec class:

```
class TmpWriterRExec(rexec.RExec):
    def r_open(self, file, mode='r', buf=-1):
        if mode in ('r', 'rb'):
            pass
        elif mode in ('w', 'wb', 'a', 'ab'):
            # check filename: must begin with /tmp/
            if file[:5]!='/tmp/':
```

(continues on next page)

(continued from previous page)

```
            raise IOError("can't write outside /tmp")
        elif (string.find(file, '/../') >= 0 or
              file[:3] == '../' or file[-3:] == '/..'):
            raise IOError("'..' in filename forbidden")
    else: raise IOError("Illegal open() mode")
    return open(file, mode, buf)
```

Notice that the above code will occasionally forbid a perfectly valid filename; for example, code in the restricted environment won't be able to open a file called /tmp/foo/../bar. To fix this, the r_open() method would have to simplify the filename to /tmp/bar, which would require splitting apart the filename and performing various operations on it. In cases where security is at stake, it may be preferable to write simple code which is sometimes overly restrictive, instead of more general code that is also more complex and may harbor a subtle security hole.

30.2 Bastion — Restricting access to objects

Deprecated since version 2.6: The Bastion module has been removed in Python 3.

Changed in version 2.3: Disabled module.

Note: The documentation has been left in place to help in reading old code that uses the module.

According to the dictionary, a bastion is "a fortified area or position", or "something that is considered a stronghold." It's a suitable name for this module, which provides a way to forbid access to certain attributes of an object. It must always be used with the rexec module, in order to allow restricted-mode programs access to certain safe attributes of an object, while denying access to other, unsafe attributes.

Bastion.Bastion(object[, filter[, name[, class]]])
: Protect the object object, returning a bastion for the object. Any attempt to access one of the object's attributes will have to be approved by the filter function; if the access is denied an AttributeError exception will be raised.

  If present, filter must be a function that accepts a string containing an attribute name, and returns true if access to that attribute will be permitted; if filter returns false, the access is denied. The default filter denies access to any function beginning with an underscore (`'_'`). The bastion's string representation will be <Bastion for name> if a value for name is provided; otherwise, repr(object) will be used.

  class, if present, should be a subclass of BastionClass; see the code in bastion.py for the details. Overriding the default BastionClass will rarely be required.

class Bastion.BastionClass(getfunc, name)
: Class which actually implements bastion objects. This is the default class used by Bastion(). The getfunc parameter is a function which returns the value of an attribute which should be exposed to the restricted execution environment when called with the name of the attribute as the only parameter. name is used to construct the repr() of the BastionClass instance.

See also:

Grail Home Page Grail, an Internet browser written in Python, uses these modules to support Python applets. More information on the use of Python's restricted execution mode in Grail is available on the Web site.

CHAPTER

THIRTYONE

IMPORTING MODULES

The modules described in this chapter provide new ways to import other Python modules and hooks for customizing the import process.

The full list of modules described in this chapter is:

31.1 imp — Access the import internals

This module provides an interface to the mechanisms used to implement the import statement. It defines the following constants and functions:

imp.get_magic()
: Return the magic string value used to recognize byte-compiled code files (.pyc files). (This value may be different for each Python version.)

imp.get_suffixes()
: Return a list of 3-element tuples, each describing a particular type of module. Each triple has the form (suffix, mode, type), where suffix is a string to be appended to the module name to form the filename to search for, mode is the mode string to pass to the built-in open() function to open the file (this can be `'r'` for text files or `'rb'` for binary files), and type is the file type, which has one of the values PY_SOURCE, PY_COMPILED, or C_EXTENSION, described below.

imp.find_module(name[, path])
: Try to find the module name. If path is omitted or None, the list of directory names given by sys.path is searched, but first a few special places are searched: the function tries to find a built-in module with the given name (C_BUILTIN), then a frozen module (PY_FROZEN), and on some systems some other places are looked in as well (on Windows, it looks in the registry which may point to a specific file).

  Otherwise, path must be a list of directory names; each directory is searched for files with any of the suffixes returned by get_suffixes() above. Invalid names in the list are silently ignored (but all list items must be strings).

  If search is successful, the return value is a 3-element tuple (file, pathname, description):

  file is an open file object positioned at the beginning, pathname is the pathname of the file found, and description is a 3-element tuple as contained in the list returned by get_suffixes() describing the kind of module found.

  If the module does not live in a file, the returned file is None, pathname is the empty string, and the description tuple contains empty strings for its suffix and mode; the module type is indicated as given in parentheses above. If the search is unsuccessful, ImportError is raised. Other exceptions indicate problems with the arguments or environment.

If the module is a package, file is None, pathname is the package path and the last item in the description tuple is PKG_DIRECTORY.

This function does not handle hierarchical module names (names containing dots). In order to find P.M, that is, submodule M of package P, use find_module() and load_module() to find and load package P, and then use find_module() with the path argument set to P.__path__. When P itself has a dotted name, apply this recipe recursively.

imp.load_module(name, file, pathname, description)
Load a module that was previously found by find_module() (or by an otherwise conducted search yielding compatible results). This function does more than importing the module: if the module was already imported, it is equivalent to a reload()! The name argument indicates the full module name (including the package name, if this is a submodule of a package). The file argument is an open file, and pathname is the corresponding file name; these can be None and '', respectively, when the module is a package or not being loaded from a file. The description argument is a tuple, as would be returned by get_suffixes(), describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually ImportError) is raised.

Important: the caller is responsible for closing the file argument, if it was not None, even when an exception is raised. This is best done using a try ... finally statement.

imp.new_module(name)
Return a new empty module object called name. This object is not inserted in sys.modules.

imp.lock_held()
Return True if the import lock is currently held, else False. On platforms without threads, always return False.

On platforms with threads, a thread executing an import holds an internal lock until the import is complete. This lock blocks other threads from doing an import until the original import completes, which in turn prevents other threads from seeing incomplete module objects constructed by the original thread while in the process of completing its import (and the imports, if any, triggered by that).

imp.acquire_lock()
Acquire the interpreter's import lock for the current thread. This lock should be used by import hooks to ensure thread-safety when importing modules.

Once a thread has acquired the import lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

On platforms without threads, this function does nothing.

New in version 2.3.

imp.release_lock()
Release the interpreter's import lock. On platforms without threads, this function does nothing.

New in version 2.3.

The following constants with integer values, defined in this module, are used to indicate the search result of find_module().

imp.PY_SOURCE
The module was found as a source file.

imp.PY_COMPILED
The module was found as a compiled code object file.

imp.C_EXTENSION
The module was found as dynamically loadable shared library.

imp.PKG_DIRECTORY
The module was found as a package directory.

imp.C_BUILTIN
The module was found as a built-in module.

imp.PY_FROZEN
The module was found as a frozen module (see init_frozen()).

The following constant and functions are obsolete; their functionality is available through find_module() or load_module(). They are kept around for backward compatibility:

imp.SEARCH_ERROR
Unused.

imp.init_builtin(name)
Initialize the built-in module called name and return its module object along with storing it in sys.modules. If the module was already initialized, it will be initialized again. Re-initialization involves the copying of the built-in module's __dict__ from the cached module over the module's entry in sys.modules. If there is no built-in module called name, None is returned.

imp.init_frozen(name)
Initialize the frozen module called name and return its module object. If the module was already initialized, it will be initialized again. If there is no frozen module called name, None is returned. (Frozen modules are modules written in Python whose compiled byte-code object is incorporated into a custom-built Python interpreter by Python's freeze utility. See Tools/freeze/ for now.)

imp.is_builtin(name)
Return 1 if there is a built-in module called name which can be initialized again. Return -1 if there is a built-in module called name which cannot be initialized again (see init_builtin()). Return 0 if there is no built-in module called name.

imp.is_frozen(name)
Return True if there is a frozen module (see init_frozen()) called name, or False if there is no such module.

imp.load_compiled(name, pathname[, file])
Load and initialize a module implemented as a byte-compiled code file and return its module object. If the module was already initialized, it will be initialized again. The name argument is used to create or access a module object. The pathname argument points to the byte-compiled code file. The file argument is the byte-compiled code file, open for reading in binary mode, from the beginning. It must currently be a real file object, not a user-defined class emulating a file.

imp.load_dynamic(name, pathname[, file])
Load and initialize a module implemented as a dynamically loadable shared library and return its module object. If the module was already initialized, it will be initialized again. Re-initialization involves copying the __dict__ attribute of the cached instance of the module over the value used in the module cached in sys.modules. The pathname argument must point to the shared library. The name argument is used to construct the name of the initialization function: an external C function called initname() in the shared library is called. The optional file argument is ignored. (Note: using shared libraries is highly system dependent, and not all systems support it.)

CPython implementation detail: The import internals identify extension modules by filename, so doing foo = load_dynamic("foo", "mod.so") and bar = load_dynamic("bar", "mod.so") will result in both foo and bar referring to the same module, regardless of whether or not mod.so exports an initbar function. On systems which support them, symlinks can be used to import multiple modules from the same shared library, as each reference to the module will use a different file name.

imp.load_source(name, pathname[, file])
Load and initialize a module implemented as a Python source file and return its module object. If the

module was already initialized, it will be initialized again. The name argument is used to create or access a module object. The pathname argument points to the source file. The file argument is the source file, open for reading as text, from the beginning. It must currently be a real file object, not a user-defined class emulating a file. Note that if a properly matching byte-compiled file (with suffix .pyc or .pyo) exists, it will be used instead of parsing the given source file.

class imp.NullImporter(path_string)
The NullImporter type is a PEP 302 import hook that handles non-directory path strings by failing to find any modules. Calling this type with an existing directory or empty string raises ImportError. Otherwise, a NullImporter instance is returned.

Python adds instances of this type to sys.path_importer_cache for any path entries that are not directories and are not handled by any other path hooks on sys.path_hooks. Instances have only one method:

find_module(fullname[, path])
This method always returns None, indicating that the requested module could not be found.

New in version 2.5.

31.1.1 Examples

The following function emulates what was the standard import statement up to Python 1.4 (no hierarchical module names). (This implementation wouldn't work in that version, since find_module() has been extended and load_module() has been added in 1.4.)

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.

    fp, pathname, description = imp.find_module(name)

    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()
```

A more complete example that implements hierarchical module names and includes a reload() function can be found in the module knee. The knee module can be found in Demo/imputil/ in the Python source distribution.

31.2 importlib — Convenience wrappers for __import__()

New in version 2.7.

This module is a minor subset of what is available in the more full-featured package of the same name from Python 3.1 that provides a complete implementation of import. What is here has been provided to help ease in transitioning from 2.7 to 3.1.

importlib.import_module(name, package=None)
: Import a module. The name argument specifies what module to import in absolute or relative terms (e.g. either pkg.mod or ..mod). If the name is specified in relative terms, then the package argument must be specified to the package which is to act as the anchor for resolving the package name (e.g. import_module('..mod', 'pkg.subpkg') will import pkg.mod). The specified module will be inserted into sys.modules and returned.

31.3 imputil — Import utilities

Deprecated since version 2.6: The imputil module has been removed in Python 3.

This module provides a very handy and useful mechanism for custom import hooks. Compared to the older ihooks module, imputil takes a dramatically simpler and more straight-forward approach to custom import functions.

class imputil.ImportManager([fs_imp])
: Manage the import process.

  install([namespace])
  : Install this ImportManager into the specified namespace.

  uninstall()
  : Restore the previous import mechanism.

  add_suffix(suffix, importFunc)
  : Undocumented.

class imputil.Importer
: Base class for replacing standard import functions.

  import_top(name)
  : Import a top-level module.

  get_code(parent, modname, fqname)
  : Find and retrieve the code for the given module.

    parent specifies a parent module to define a context for importing. It may be None, indicating no particular context for the search.

    modname specifies a single module (not dotted) within the parent.

    fqname specifies the fully-qualified module name. This is a (potentially) dotted name from the "root" of the module namespace down to the modname.

    If there is no parent, then modname==fqname.

    This method should return None, or a 3-tuple.

    - If the module was not found, then None should be returned.
    - The first item of the 2- or 3-tuple should be the integer 0 or 1, specifying whether the module that was found is a package or not.
    - The second item is the code object for the module (it will be executed within the new module's namespace). This item can also be a fully-loaded module object (e.g. loaded from a shared lib).

- The third item is a dictionary of name/value pairs that will be inserted into new module before the code object is executed. This is provided in case the module's code expects certain values (such as where the module was found). When the second item is a module object, then these names/values will be inserted after the module has been loaded/initialized.

class imputil.BuiltinImporter
Emulate the import mechanism for built-in and frozen modules. This is a sub-class of the Importer class.

get_code(parent, modname, fqname)
Undocumented.

imputil.py_suffix_importer(filename, finfo, fqname)
Undocumented.

class imputil.DynLoadSuffixImporter([desc])
Undocumented.

import_file(filename, finfo, fqname)
Undocumented.

31.3.1 Examples

This is a re-implementation of hierarchical module import.

This code is intended to be read, not executed. However, it does work – all you need to do to enable it is "import knee".

(The name is a pun on the clunkier predecessor of this module, "ni".)

```
import sys, imp, __builtin__

# Replacement for __import__()
def import_hook(name, globals=None, locals=None, fromlist=None):
    parent = determine_parent(globals)
    q, tail = find_head_package(parent, name)
    m = load_tail(q, tail)
    if not fromlist:
        return q
    if hasattr(m, "__path__"):
        ensure_fromlist(m, fromlist)
    return m

def determine_parent(globals):
    if not globals or  not globals.has_key("__name__"):
        return None
    pname = globals['__name__']
    if globals.has_key("__path__"):
        parent = sys.modules[pname]
        assert globals is parent.__dict__
        return parent
    if '.' in pname:
        i = pname.rfind('.')
        pname = pname[:i]
        parent = sys.modules[pname]
        assert parent.__name__ == pname
        return parent
    return None
```

(continues on next page)

(continued from previous page)

```
def find_head_package(parent, name):
    if '.' in name:
        i = name.find('.')
        head = name[:i]
        tail = name[i+1:]
    else:
        head = name
        tail = ""
    if parent:
        qname = "%s.%s" % (parent.__name__, head)
    else:
        qname = head
    q = import_module(head, qname, parent)
    if q: return q, tail
    if parent:
        qname = head
        parent = None
        q = import_module(head, qname, parent)
        if q: return q, tail
    raise ImportError("No module named " + qname)

def load_tail(q, tail):
    m = q
    while tail:
        i = tail.find('.')
        if i < 0: i = len(tail)
        head, tail = tail[:i], tail[i+1:]
        mname = "%s.%s" % (m.__name__, head)
        m = import_module(head, mname, m)
        if not m:
            raise ImportError("No module named " + mname)
    return m

def ensure_fromlist(m, fromlist, recursive=0):
    for sub in fromlist:
        if sub == "*":
            if not recursive:
                try:
                    all = m.__all__
                except AttributeError:
                    pass
                else:
                    ensure_fromlist(m, all, 1)
            continue
        if sub != "*" and not hasattr(m, sub):
            subname = "%s.%s" % (m.__name__, sub)
            submod = import_module(sub, subname, m)
            if not submod:
                raise ImportError("No module named " + subname)

def import_module(partname, fqname, parent):
    try:
        return sys.modules[fqname]
    except KeyError:
        pass
```

(continues on next page)

(continued from previous page)

```
    try:
        fp, pathname, stuff = imp.find_module(partname,
                                              parent and parent.__path__)
    except ImportError:
        return None
    try:
        m = imp.load_module(fqname, fp, pathname, stuff)
    finally:
        if fp: fp.close()
    if parent:
        setattr(parent, partname, m)
    return m


# Replacement for reload()
def reload_hook(module):
    name = module.__name__
    if '.' not in name:
        return import_module(name, name, None)
    i = name.rfind('.')
    pname = name[:i]
    parent = sys.modules[pname]
    return import_module(name[i+1:], name, parent)


# Save the original hooks
original_import = __builtin__.__import__
original_reload = __builtin__.reload

# Now install our hooks
__builtin__.__import__ = import_hook
__builtin__.reload = reload_hook
```

Also see the importers module (which can be found in Demo/imputil/ in the Python source distribution) for additional examples.

31.4 zipimport — Import modules from Zip archives

New in version 2.3.

This module adds the ability to import Python modules (*.py, *.py[co]) and packages from ZIP-format archives. It is usually not needed to use the zipimport module explicitly; it is automatically used by the built-in import mechanism for sys.path items that are paths to ZIP archives.

Typically, sys.path is a list of directory names as strings. This module also allows an item of sys.path to be a string naming a ZIP file archive. The ZIP archive can contain a subdirectory structure to support package imports, and a path within the archive can be specified to only import from a subdirectory. For example, the path example.zip/lib/ would only import from the lib/ subdirectory within the archive.

Any files may be present in the ZIP archive, but only files .py and .py[co] are available for import. ZIP import of dynamic modules (.pyd, .so) is disallowed. Note that if an archive only contains .py files, Python will not attempt to modify the archive by adding the corresponding .pyc or .pyo file, meaning that if a ZIP archive doesn't contain .pyc files, importing may be rather slow.

Using the built-in reload() function will fail if called on a module loaded from a ZIP archive; it is unlikely that reload() would be needed, since this would imply that the ZIP has been altered during runtime.

ZIP archives with an archive comment are currently not supported.

See also:

PKZIP Application Note Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

PEP 273 - Import Modules from Zip Archives Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in PEP 273, but uses an implementation written by Just van Rossum that uses the import hooks described in PEP 302.

PEP 302 - New Import Hooks The PEP to add the import hooks that help this module work.

This module defines an exception:

exception zipimport.ZipImportError
Exception raised by zipimporter objects. It's a subclass of ImportError, so it can be caught as ImportError, too.

31.4.1 zipimporter Objects

zipimporter is the class for importing ZIP files.

class zipimport.zipimporter(archivepath)
Create a new zipimporter instance. archivepath must be a path to a ZIP file, or to a specific path within a ZIP file. For example, an archivepath of foo/bar.zip/lib will look for modules in the lib directory inside the ZIP file foo/bar.zip (provided that it exists).

ZipImportError is raised if archivepath doesn't point to a valid ZIP archive.

find_module(fullname[, path])
Search for a module specified by fullname. fullname must be the fully qualified (dotted) module name. It returns the zipimporter instance itself if the module was found, or None if it wasn't. The optional path argument is ignored—it's there for compatibility with the importer protocol.

get_code(fullname)
Return the code object for the specified module. Raise ZipImportError if the module couldn't be found.

get_data(pathname)
Return the data associated with pathname. Raise IOError if the file wasn't found.

get_filename(fullname)
Return the value __file__ would be set to if the specified module was imported. Raise ZipImportError if the module couldn't be found.

New in version 2.7.

get_source(fullname)
Return the source code for the specified module. Raise ZipImportError if the module couldn't be found, return None if the archive does contain the module, but has no source for it.

is_package(fullname)
Return True if the module specified by fullname is a package. Raise ZipImportError if the module couldn't be found.

load_module(fullname)
Load the module specified by fullname. fullname must be the fully qualified (dotted) module name. It returns the imported module, or raises ZipImportError if it wasn't found.

archive
: The file name of the importer's associated ZIP file, without a possible subpath.

prefix
: The subpath within the ZIP file where modules are searched. This is the empty string for zipimporter objects which point to the root of the ZIP file.

The archive and prefix attributes, when combined with a slash, equal the original archivepath argument given to the zipimporter constructor.

31.4.2 Examples

Here is an example that imports a module from a ZIP archive - note that the zipimport module is not explicitly used.

```
$ unzip -l example.zip
Archive:  example.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                     1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

31.5 pkgutil — Package extension utility

New in version 2.3.

Source code: Lib/pkgutil.py

This module provides utilities for the import system, in particular package support.

pkgutil.extend_path(path, name)
: Extend the search path for the modules which comprise a package. Intended use is to place the following code in a package's __init__.py:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

This will add to the package's __path__ all subdirectories of directories on sys.path named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for *.pkg files beginning where * matches the name argument. This feature is similar to *.pth files (see the site module for more information), except that it doesn't special-case lines starting with import. A *.pkg file is trusted at face value: apart from checking for duplicates, all entries found in a *.pkg file are added to the path, regardless of whether they exist on the filesystem. (This is a feature.)

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that sys.path is a sequence. Items of sys.path that are not (Unicode or 8-bit) strings referring to existing directories are ignored. Unicode items on sys.path that cause errors when used as filenames may cause this function to raise an exception (in line with os.path.isdir() behavior).

class pkgutil.ImpImporter(dirname=None)
PEP 302 Importer that wraps Python's "classic" import algorithm.

If dirname is a string, a PEP 302 importer is created that searches that directory. If dirname is None, a PEP 302 importer is created that searches the current sys.path, plus any modules that are frozen or built-in.

Note that ImpImporter does not currently support being used by placement on sys.meta_path.

class pkgutil.ImpLoader(fullname, file, filename, etc)
PEP 302 Loader that wraps Python's "classic" import algorithm.

pkgutil.find_loader(fullname)
Find a PEP 302 "loader" object for fullname.

If fullname contains dots, path must be the containing package's __path__. Returns None if the module cannot be found or imported. This function uses iter_importers(), and is thus subject to the same limitations regarding platform-specific special import locations such as the Windows registry.

pkgutil.get_importer(path_item)
Retrieve a PEP 302 importer for the given path_item.

The returned importer is cached in sys.path_importer_cache if it was newly created by a path hook.

If there is no importer, a wrapper around the basic import machinery is returned. This wrapper is never inserted into the importer cache (None is inserted instead).

The cache (or part of it) can be cleared manually if a rescan of sys.path_hooks is necessary.

pkgutil.get_loader(module_or_name)
Get a PEP 302 "loader" object for module_or_name.

If the module or package is accessible via the normal import mechanism, a wrapper around the relevant part of that machinery is returned. Returns None if the module cannot be found or imported. If the named module is not already imported, its containing package (if any) is imported, in order to establish the package __path__.

This function uses iter_importers(), and is thus subject to the same limitations regarding platform-specific special import locations such as the Windows registry.

pkgutil.iter_importers(fullname='')
Yield PEP 302 importers for the given module name.

If fullname contains a '.', the importers will be for the package containing fullname, otherwise they will be importers for sys.meta_path, sys.path, and Python's "classic" import machinery, in that order. If the named module is in a package, that package is imported as a side effect of invoking this function.

Non-PEP 302 mechanisms (e.g. the Windows registry) used by the standard import machinery to find files in alternative locations are partially supported, but are searched after sys.path. Normally, these locations are searched before sys.path, preventing sys.path entries from shadowing them.

For this to cause a visible difference in behaviour, there must be a module or package name that is accessible via both sys.path and one of the non-PEP 302 file system mechanisms. In this case, the emulation will find the former version, while the builtin import mechanism will find the latter.

Items of the following types can be affected by this discrepancy: imp.C_EXTENSION, imp.PY_SOURCE, imp.PY_COMPILED, imp.PKG_DIRECTORY.

pkgutil.iter_modules(path=None, prefix='')
Yields (module_loader, name, ispkg) for all submodules on path, or, if path is None, all top-level modules on sys.path.

path should be either None or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

pkgutil.walk_packages(path=None, prefix='', onerror=None)
Yields (module_loader, name, ispkg) for all modules recursively on path, or, if path is None, all accessible modules.

path should be either None or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note that this function must import all packages (not all modules!) on the given path, in order to access the __path__ attribute to find submodules.

onerror is a function which gets called with one argument (the name of the package which was being imported) if any exception occurs while trying to import a package. If no onerror function is supplied, ImportErrors are caught and ignored, while all other exceptions are propagated, terminating the search.

Examples:

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

pkgutil.get_data(package, resource)
Get a resource from a package.

This is a wrapper for the PEP 302 loader get_data() API. The package argument should be the name of a package, in standard module format (foo.bar). The resource argument should be in the form of a relative filename, using / as the path separator. The parent directory name .. is not allowed, and nor is a rooted name (starting with a /).

The function returns a binary string that is the contents of the specified resource.

For packages located in the filesystem, which have already been imported, this is the rough equivalent of:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

If the package cannot be located or loaded, or it uses a PEP 302 loader which does not support get_data(), then None is returned.

New in version 2.6.

31.6 modulefinder — Find modules used by a script

New in version 2.3.

Source code: Lib/modulefinder.py

This module provides a ModuleFinder class that can be used to determine the set of modules imported by a script. modulefinder.py can also be run as a script, giving the filename of a Python script as its argument, after which a report of the imported modules will be printed.

modulefinder.AddPackagePath(pkg_name, path)
Record that the package named pkg_name can be found in the specified path.

modulefinder.ReplacePackage(oldname, newname)
Allows specifying that the module named oldname is in fact the package named newname. The most common usage would be to handle how the _xmlplus package replaces the xml package.

class modulefinder.ModuleFinder([path=None, debug=0, excludes=[], replace_paths=[]])
This class provides run_script() and report() methods to determine the set of modules imported by a script. path can be a list of directories to search for modules; if not specified, sys.path is used. debug sets the debugging level; higher values make the class print debugging messages about what it's doing. excludes is a list of module names to exclude from the analysis. replace_paths is a list of (oldpath, newpath) tuples that will be replaced in module paths.

report()
Print a report to standard output that lists the modules imported by the script and their paths, as well as modules that are missing or seem to be missing.

run_script(pathname)
Analyze the contents of the pathname file, which must contain Python code.

modules
A dictionary mapping module names to modules. See Example usage of ModuleFinder.

31.6.1 Example usage of ModuleFinder

The script that is going to get analyzed later on (bacon.py):

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

The script that will output the report of bacon.py:

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print 'Loaded modules:'
for name, mod in finder.modules.iteritems():
    print '%s: ' % name,
    print ','.join(mod.globalnames.keys()[:3])
```

(continues on next page)

(continued from previous page)

```
print '-'*50
print 'Modules not imported:'
print '\n'.join(finder.badmodules.iterkeys())
```

Sample output (may vary depending on the architecture):

```
Loaded modules:
_types:
copy_reg:  _inverted_registry,_slotnames,__all__
sre_compile:  isstring,_sre,_optimize_unicode
_sre:
sre_constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
sre_parse:  __getslice__,_PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
--------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

31.7 runpy — Locating and executing Python modules

New in version 2.5.

Source code: Lib/runpy.py

The runpy module is used to locate and run Python modules without importing them first. Its main use is to implement the -m command line switch that allows scripts to be located using the Python module namespace rather than the filesystem.

The runpy module provides two functions:

runpy.run_module(mod_name, init_globals=None, run_name=None, alter_sys=False)

Execute the code of the specified module and return the resulting module globals dictionary. The module's code is first located using the standard import mechanism (refer to PEP 302 for details) and then executed in a fresh module namespace.

If the supplied module name refers to a package rather than a normal module, then that package is imported and the __main__ submodule within that package is then executed and the resulting module globals dictionary returned.

The optional dictionary argument init_globals may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by run_module().

The special global variables __name__, __file__, __loader__ and __package__ are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

__name__ is set to run_name if this optional argument is not None, to mod_name + '.__main__' if the named module is a package and to the mod_name argument otherwise.

__file__ is set to the name provided by the module loader. If the loader does not make filename information available, this variable is set to None.

__loader__ is set to the PEP 302 module loader used to retrieve the code for the module (This loader may be a wrapper around the standard import mechanism).

__package__ is set to mod_name if the named module is a package and to mod_name.rpartition('.')[0] otherwise.

If the argument alter_sys is supplied and evaluates to True, then sys.argv[0] is updated with the value of __file__ and sys.modules[__name__] is updated with a temporary module object for the module being executed. Both sys.argv[0] and sys.modules[__name__] are restored to their original values before the function returns.

Note that this manipulation of sys is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the sys module be left alone when invoking this function from threaded code.

See also:

The -m option offering equivalent functionality from the command line.

Changed in version 2.7: Added ability to execute packages by looking for a __main__ submodule

runpy.run_path(file_path, init_globals=None, run_name=None)

Execute the code at the named filesystem location and return the resulting module globals dictionary. As with a script name supplied to the CPython command line, the supplied path may refer to a Python source file, a compiled bytecode file or a valid sys.path entry containing a __main__ module (e.g. a zipfile containing a top-level __main__.py file).

For a simple script, the specified code is simply executed in a fresh module namespace. For a valid sys.path entry (typically a zipfile or directory), the entry is first added to the beginning of sys.path. The function then looks for and executes a __main__ module using the updated path. Note that there is no special protection against invoking an existing __main__ entry located elsewhere on sys.path if there is no such module at the specified location.

The optional dictionary argument init_globals may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by run_path().

The special global variables __name__, __file__, __loader__ and __package__ are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

__name__ is set to run_name if this optional argument is not None and to '<run_path>' otherwise.

__file__ is set to the name provided by the module loader. If the loader does not make filename information available, this variable is set to None. For a simple script, this will be set to file_path.

__loader__ is set to the PEP 302 module loader used to retrieve the code for the module (This loader may be a wrapper around the standard import mechanism). For a simple script, this will be set to None.

__package__ is set to __name__.rpartition('.')[0].

A number of alterations are also made to the sys module. Firstly, sys.path may be altered as described above. sys.argv[0] is updated with the value of file_path and sys.modules[__name__] is updated

with a temporary module object for the module being executed. All modifications to items in sys are reverted before the function returns.

Note that, unlike run_module(), the alterations made to sys are not optional in this function as these adjustments are essential to allowing the execution of sys.path entries. As the thread-safety limitations still apply, use of this function in threaded code should be either serialised with the import lock or delegated to a separate process.

See also:

using-on-interface-options for equivalent functionality on the command line (python path/to/script).

New in version 2.7.

See also:

PEP 338 – Executing modules as scripts PEP written and implemented by Nick Coghlan.

PEP 366 – Main module explicit relative imports PEP written and implemented by Nick Coghlan.

using-on-general - CPython command line details

CHAPTER

THIRTYTWO

PYTHON LANGUAGE SERVICES

Python provides a number of modules to assist in working with the Python language. These modules support tokenizing, parsing, syntax analysis, bytecode disassembly, and various other facilities.

These modules include:

32.1 parser — Access Python parse trees

The parser module provides an interface to Python's internal parser and byte-code compiler. The primary purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable code from this. This is better than trying to parse and modify an arbitrary Python code fragment as a string because parsing is performed in a manner identical to the code forming the application. It is also faster.

Note: From Python 2.5 onward, it's much more convenient to cut in at the Abstract Syntax Tree (AST) generation and compilation stage, using the ast module.

The parser module exports the names documented here also with "st" replaced by "ast"; this is a legacy from the time when there was no other AST and has nothing to do with the AST found in Python 2.5. This is also the reason for the functions' keyword arguments being called ast, not st. The "ast" functions have been removed in Python 3.

There are a few things to note about this module which are important to making use of the data structures created. This is not a tutorial on editing the parse trees for Python code, but some examples of using the parser module are presented.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to reference-index. The parser itself is created from a grammar specification defined in the file Grammar/Grammar in the standard Python distribution. The parse trees stored in the ST objects created by this module are the actual output from the internal parser when created by the expr() or suite() functions, described below. The ST objects created by sequence2st() faithfully simulate those structures. Be aware that the values of the sequences which are considered "correct" will vary from one version of Python to another as the formal grammar for the language is revised. However, transporting code from one Python version to another as source text will always allow correct parse trees to be created in the target version, with the only restriction being that migrating to an older version of the interpreter will not support more recent language constructs. The parse trees are not typically compatible from one version to another, whereas source code has always been forward-compatible.

Each element of the sequences returned by st2list() or st2tuple() has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file

Include/graminit.h and the Python module symbol. Each additional element of the sequence represents a component of the production as recognized in the input string: these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword if in an if_stmt, are included in the node tree without any special treatment. For example, the if keyword is represented by the tuple (1, 'if'), where 1 is the numeric value associated with all NAME tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as (1, 'if', 12), where the 12 represents the line number at which the terminal symbol was found.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the if keyword above is representative. The various types of terminal symbols are defined in the C header file Include/token.h and the Python module token.

The ST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple "wrapper" class may be created in Python to hide the use of ST objects.

The parser module defines functions for a few distinct purposes. The most important purposes are to create ST objects and to convert ST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an ST object.

See also:

Module symbol Useful constants representing internal nodes of the parse tree.

Module token Useful constants representing leaf nodes of the parse tree and functions for testing node values.

32.1.1 Creating ST Objects

ST objects may be created from source code or from a parse tree. When creating an ST object from source, different functions are used to create the 'eval' and 'exec' forms.

parser.expr(source)
: The expr() function parses the parameter source as if it were an input to compile(source, 'file.py', 'eval'). If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

parser.suite(source)
: The suite() function parses the parameter source as if it were an input to compile(source, 'file.py', 'exec'). If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

parser.sequence2st(sequence)
: This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an ST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a ParserError exception is raised. An ST object created this way should not be assumed to compile correctly; normal exceptions raised by compilation may still be initiated when the ST object is passed to compilest(). This may indicate problems not related to syntax (such as a MemoryError exception), but may also be due to constructs such as the result of parsing del f(0), which escapes the Python parser but is checked by the bytecode compiler.

  Sequences representing terminal tokens may be represented as either two-element lists of the form (1, 'name') or as three-element lists of the form (1, 'name', 56). If the third element is present, it is

assumed to be a valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

parser.tuple2st(sequence)
This is the same function as sequence2st(). This entry point is maintained for backward compatibility.

32.1.2 Converting ST Objects

ST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple- trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

parser.st2list(ast[, line_info])
This function accepts an ST object from the caller in ast and returns a Python list representing the equivalent parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, st2tuple() should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If line_info is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token ends. This information is omitted if the flag is false or omitted.

parser.st2tuple(ast[, line_info])
This function accepts an ST object from the caller in ast and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to st2list().

If line_info is true, line number information will be included for all terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.

parser.compilest(ast, filename='<syntax-tree>')
The Python byte compiler can be invoked on an ST object to produce code objects which can be used as part of an exec statement or a call to the built-in eval() function. This function provides the interface to the compiler, passing the internal parse tree from ast to the parser, using the source file name specified by the filename parameter. The default value supplied for filename indicates that the source was an ST object.

Compiling an ST object may result in exceptions related to compilation; an example would be a SyntaxError caused by the parse tree for del f(0): this statement is considered legal within the formal grammar for Python but is not a legal language construct. The SyntaxError raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the parser module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.

32.1.3 Queries on ST Objects

Two functions are provided which allow an application to determine if an ST was created as an expression or a suite. Neither of these functions can be used to determine if an ST was created from source code via expr() or suite() or from a parse tree via sequence2st().

parser.isexpr(ast)
When ast represents an 'eval' form, this function returns true, otherwise it returns false. This is useful, since code objects normally cannot be queried for this information using existing built-in functions.

Note that the code objects created by compilest() cannot be queried like this either, and are identical to those created by the built-in compile() function.

parser.issuite(ast)
: This function mirrors isexpr() in that it reports whether an ST object represents an `'exec'` form, commonly known as a "suite." It is not safe to assume that this function is equivalent to not isexpr(ast), as additional syntactic fragments may be supported in the future.

32.1.4 Exceptions and Error Handling

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

exception parser.ParserError
: Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built-in SyntaxError raised during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to sequence2st() and an explanatory string. Calls to sequence2st() need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions compilest(), expr(), and suite() may raise exceptions which are normally raised by the parsing and compilation process. These include the built in exceptions MemoryError, OverflowError, SyntaxError, and SystemError. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.

32.1.5 ST Objects

Ordered and equality comparisons are supported between ST objects. Pickling of ST objects (using the pickle module) is also supported.

parser.STType
: The type of the objects returned by expr(), suite() and sequence2st().

ST objects have the following methods:

ST.compile([filename])
: Same as compilest(st, filename).

ST.isexpr()
: Same as isexpr(st).

ST.issuite()
: Same as issuite(st).

ST.tolist([line_info])
: Same as st2list(st, line_info).

ST.totuple([line_info])
: Same as st2tuple(st, line_info).

32.1.6 Example: Emulation of compile()

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the parser module to produce an intermediate data structure is equivalent to the code

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the parser module is somewhat longer, and allows the intermediate internal parse tree to be retained as an ST object:

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both ST and code objects can package this code into readily available functions:

```
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```

32.2 ast — Abstract Syntax Trees

New in version 2.5: The low-level _ast module containing only the node classes.

New in version 2.6: The high-level ast module containing all helpers.

Source code: Lib/ast.py

The ast module helps Python applications to process trees of the Python abstract syntax grammar. The abstract syntax itself might change with each Python release; this module helps to find out programmatically what the current grammar looks like.

An abstract syntax tree can be generated by passing ast.PyCF_ONLY_AST as a flag to the compile() built-in function, or using the parse() helper provided in this module. The result will be a tree of objects whose classes all inherit from ast.AST. An abstract syntax tree can be compiled into a Python code object using the built-in compile() function.

32.2.1 Node classes

class ast.AST

This is the base of all AST node classes. The actual node classes are derived from the Parser/Python.asdl file, which is reproduced below. They are defined in the _ast C module and re-exported in ast.

There is one class defined for each left-hand side symbol in the abstract grammar (for example, ast.stmt or ast.expr). In addition, there is one class defined for each constructor on the right-hand side;

these classes inherit from the classes for the left-hand side trees. For example, ast.BinOp inherits from ast.expr. For production rules with alternatives (aka "sums"), the left-hand side class is abstract: only instances of specific constructor nodes are ever created.

_fields
: Each concrete class has an attribute _fields which gives the names of all child nodes.

  Each instance of a concrete class has one attribute for each child node, of the type as defined in the grammar. For example, ast.BinOp instances have an attribute left of type ast.expr.

  If these attributes are marked as optional in the grammar (using a question mark), the value might be None. If the attributes can have zero-or-more values (marked with an asterisk), the values are represented as Python lists. All possible attributes must be present and have valid values when compiling an AST with compile().

lineno
col_offset
: Instances of ast.expr and ast.stmt subclasses have lineno and col_offset attributes. The lineno is the line number of source text (1-indexed so the first line is line 1) and the col_offset is the UTF-8 byte offset of the first token that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

The constructor of a class ast.T parses its arguments as follows:

- If there are positional arguments, there must be as many as there are items in T._fields; they will be assigned as attributes of these names.
- If there are keyword arguments, they will set the attributes of the same names to the given values.

For example, to create and populate an ast.UnaryOp node, you could use

```
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Num()
node.operand.n = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```

or the more compact

```
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

New in version 2.6: The constructor as explained above was added. In Python 2.5 nodes had to be created by calling the class constructor without arguments and setting the attributes afterwards.

32.2.2 Abstract Grammar

The module defines a string constant __version__ which is the decimal Subversion revision number of the file shown below.

The abstract grammar is currently defined as follows:

```
-- ASDL's five builtin types are identifier, int, string, object, bool

module Python version "$Revision$"
```

(continues on next page)

(continued from previous page)

```
{
      mod = Module(stmt* body)
          | Interactive(stmt* body)
          | Expression(expr body)

          -- not really an actual node but useful in Jython's typesystem.
          | Suite(stmt* body)

      stmt = FunctionDef(identifier name, arguments args,
                         stmt* body, expr* decorator_list)
           | ClassDef(identifier name, expr* bases, stmt* body, expr* decorator_list)
           | Return(expr? value)

           | Delete(expr* targets)
           | Assign(expr* targets, expr value)
           | AugAssign(expr target, operator op, expr value)

           -- not sure if bool is allowed, can always use int
           | Print(expr? dest, expr* values, bool nl)

           -- use 'orelse' because else is a keyword in target languages
           | For(expr target, expr iter, stmt* body, stmt* orelse)
           | While(expr test, stmt* body, stmt* orelse)
           | If(expr test, stmt* body, stmt* orelse)
           | With(expr context_expr, expr? optional_vars, stmt* body)

           -- 'type' is a bad name
           | Raise(expr? type, expr? inst, expr? tback)
           | TryExcept(stmt* body, excepthandler* handlers, stmt* orelse)
           | TryFinally(stmt* body, stmt* finalbody)
           | Assert(expr test, expr? msg)

           | Import(alias* names)
           | ImportFrom(identifier? module, alias* names, int? level)

           -- Doesn't capture requirement that locals must be
           -- defined if globals is
           -- still supports use as a function!
           | Exec(expr body, expr? globals, expr? locals)

           | Global(identifier* names)
           | Expr(expr value)
           | Pass | Break | Continue

           -- XXX Jython will be different
           -- col_offset is the byte offset in the utf8 string the parser uses
           attributes (int lineno, int col_offset)

           -- BoolOp() can use left & right?
      expr = BoolOp(boolop op, expr* values)
           | BinOp(expr left, operator op, expr right)
           | UnaryOp(unaryop op, expr operand)
           | Lambda(arguments args, expr body)
           | IfExp(expr test, expr body, expr orelse)
           | Dict(expr* keys, expr* values)
           | Set(expr* elts)
```

(continues on next page)

(continued from previous page)

```
          | ListComp(expr elt, comprehension* generators)
          | SetComp(expr elt, comprehension* generators)
          | DictComp(expr key, expr value, comprehension* generators)
          | GeneratorExp(expr elt, comprehension* generators)
          -- the grammar constrains where yield expressions can occur
          | Yield(expr? value)
          -- need sequences for compare to distinguish between
          -- x < 4 < 3 and (x < 4) < 3
          | Compare(expr left, cmpop* ops, expr* comparators)
          | Call(expr func, expr* args, keyword* keywords,
                 expr? starargs, expr? kwargs)
          | Repr(expr value)
          | Num(object n) -- a number as a PyObject.
          | Str(string s) -- need to specify raw, unicode, etc?
          -- other literals? bools?

          -- the following expression can appear in assignment context
          | Attribute(expr value, identifier attr, expr_context ctx)
          | Subscript(expr value, slice slice, expr_context ctx)
          | Name(identifier id, expr_context ctx)
          | List(expr* elts, expr_context ctx)
          | Tuple(expr* elts, expr_context ctx)

          -- col_offset is the byte offset in the utf8 string the parser uses
          attributes (int lineno, int col_offset)

    expr_context = Load | Store | Del | AugLoad | AugStore | Param

    slice = Ellipsis | Slice(expr? lower, expr? upper, expr? step)
          | ExtSlice(slice* dims)
          | Index(expr value)

    boolop = And | Or

    operator = Add | Sub | Mult | Div | Mod | Pow | LShift
             | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs)

    -- not sure what to call the first argument for raise and except
    excepthandler = ExceptHandler(expr? type, expr? name, stmt* body)
                    attributes (int lineno, int col_offset)

    arguments = (expr* args, identifier? vararg,
                 identifier? kwarg, expr* defaults)

    -- keyword arguments supplied to call
    keyword = (identifier arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)
}
```

32.2.3 ast Helpers

New in version 2.6.

Apart from the node classes, ast module defines these utility functions and classes for traversing abstract syntax trees:

ast.parse(source, filename='<unknown>', mode='exec')
Parse the source into an AST node. Equivalent to compile(source, filename, mode, ast.PyCF_ONLY_AST).

ast.literal_eval(node_or_string)
Safely evaluate an expression node or a Unicode or Latin-1 encoded string containing a Python literal or container display. The string or node provided may only consist of the following Python literal structures: strings, numbers, tuples, lists, dicts, booleans, and None.

This can be used for safely evaluating strings containing Python values from untrusted sources without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

ast.get_docstring(node, clean=True)
Return the docstring of the given node (which must be a FunctionDef, ClassDef or Module node), or None if it has no docstring. If clean is true, clean up the docstring's indentation with inspect.cleandoc().

ast.fix_missing_locations(node)
When you compile a node tree with compile(), the compiler expects lineno and col_offset attributes for every node that supports them. This is rather tedious to fill in for generated nodes, so this helper adds these attributes recursively where not already set, by setting them to the values of the parent node. It works recursively starting at node.

ast.increment_lineno(node, n=1)
Increment the line number of each node in the tree starting at node by n. This is useful to "move code" to a different location in a file.

ast.copy_location(new_node, old_node)
Copy source location (lineno and col_offset) from old_node to new_node if possible, and return new_node.

ast.iter_fields(node)
Yield a tuple of (fieldname, value) for each field in node._fields that is present on node.

ast.iter_child_nodes(node)
Yield all direct child nodes of node, that is, all fields that are nodes and all items of fields that are lists of nodes.

ast.walk(node)
Recursively yield all descendant nodes in the tree starting at node (including node itself), in no specified order. This is useful if you only want to modify nodes in place and don't care about the context.

class ast.NodeVisitor
A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the visit() method.

This class is meant to be subclassed, with the subclass adding visitor methods.

visit(node)
Visit a node. The default implementation calls the method called self.visit_classname where classname is the name of the node class, or generic_visit() if that method doesn't exist.

generic_visit(node)
This visitor calls visit() on all children of the node.

Note that child nodes of nodes that have a custom visitor method won't be visited unless the visitor calls generic_visit() or visits them itself.

Don't use the NodeVisitor if you want to apply changes to nodes during traversal. For this a special visitor exists (NodeTransformer) that allows modifications.

class ast.NodeTransformer

A NodeVisitor subclass that walks the abstract syntax tree and allows modification of nodes.

The NodeTransformer will walk the AST and use the return value of the visitor methods to replace or remove the old node. If the return value of the visitor method is None, the node will be removed from its location, otherwise it is replaced with the return value. The return value may be the original node in which case no replacement takes place.

Here is an example transformer that rewrites all occurrences of name lookups (foo) to data['foo']:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return copy_location(Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        ), node)
```

Keep in mind that if the node you're operating on has child nodes you must either transform the child nodes yourself or call the generic_visit() method for the node first.

For nodes that were part of a collection of statements (that applies to all statement nodes), the visitor may also return a list of nodes rather than just a single node.

Usually you use the transformer like this:

```
node = YourTransformer().visit(node)
```

ast.dump(node, annotate_fields=True, include_attributes=False)

Return a formatted dump of the tree in node. This is mainly useful for debugging purposes. The returned string will show the names and the values for fields. This makes the code impossible to evaluate, so if evaluation is wanted annotate_fields must be set to False. Attributes such as line numbers and column offsets are not dumped by default. If this is wanted, include_attributes can be set to True.

32.3 symtable — Access to the compiler's symbol tables

Source code: Lib/symtable.py

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. symtable provides an interface to examine these tables.

32.3.1 Generating Symbol Tables

symtable.symtable(code, filename, compile_type)

Return the toplevel SymbolTable for the Python source code. filename is the name of the file containing the code. compile_type is like the mode argument to compile().

32.3.2 Examining Symbol Tables

class symtable.SymbolTable
A namespace table for a block. The constructor is not public.

get_type()
Return the type of the symbol table. Possible values are 'class', 'module', and 'function'.

get_id()
Return the table's identifier.

get_name()
Return the table's name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or 'top' if the table is global (get_type() returns 'module').

get_lineno()
Return the number of the first line in the block this table represents.

is_optimized()
Return True if the locals in this table can be optimized.

is_nested()
Return True if the block is a nested class or function.

has_children()
Return True if the block has nested namespaces within it. These can be obtained with get_children().

has_exec()
Return True if the block uses exec.

has_import_star()
Return True if the block uses a starred from-import.

get_identifiers()
Return a list of names of symbols in this table.

lookup(name)
Lookup name in the table and return a Symbol instance.

get_symbols()
Return a list of Symbol instances for names in the table.

get_children()
Return a list of the nested symbol tables.

class symtable.Function
A namespace for a function or method. This class inherits SymbolTable.

get_parameters()
Return a tuple containing names of parameters to this function.

get_locals()
Return a tuple containing names of locals in this function.

get_globals()
Return a tuple containing names of globals in this function.

get_frees()
Return a tuple containing names of free variables in this function.

class symtable.Class
A namespace of a class. This class inherits SymbolTable.

get_methods()
Return a tuple containing the names of methods declared in the class.

class symtable.Symbol
An entry in a SymbolTable corresponding to an identifier in the source. The constructor is not public.

get_name()
Return the symbol's name.

is_referenced()
Return True if the symbol is used in its block.

is_imported()
Return True if the symbol is created from an import statement.

is_parameter()
Return True if the symbol is a parameter.

is_global()
Return True if the symbol is global.

is_declared_global()
Return True if the symbol is declared global with a global statement.

is_local()
Return True if the symbol is local to its block.

is_free()
Return True if the symbol is referenced in its block, but not assigned to.

is_assigned()
Return True if the symbol is assigned to in its block.

is_namespace()
Return True if name binding introduces new namespace.

If the name is used as the target of a function or class statement, this will be true.

For example:

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

Note that a single name can be bound to multiple objects. If the result is True, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

get_namespaces()
Return a list of namespaces bound to this name.

get_namespace()
Return the namespace bound to this name. If more than one namespace is bound, a ValueError is raised.

32.4 symbol — Constants used with Python parse trees

Source code: Lib/symbol.py

This module provides constants which represent the numeric values of internal nodes of the parse tree. Unlike most Python constants, these use lower-case names. Refer to the file Grammar/Grammar in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

This module also provides one additional data object:

symbol.sym_name
: Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

32.5 token — Constants used with Python parse trees

Source code: Lib/token.py

This module provides constants which represent the numeric values of leaf nodes of the parse tree (terminal tokens). Refer to the file Grammar/Grammar in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

The module also provides a mapping from numeric codes to names and some functions. The functions mirror definitions in the Python C header files.

token.tok_name
: Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

token.ISTERMINAL(x)
: Return true for terminal token values.

token.ISNONTERMINAL(x)
: Return true for non-terminal token values.

token.ISEOF(x)
: Return true if x is the marker indicating the end of input.

The token constants are:

token.ENDMARKER
token.NAME
token.NUMBER
token.STRING
token.NEWLINE
token.INDENT
token.DEDENT
token.LPAR
token.RPAR
token.LSQB
token.RSQB
token.COLON
token.COMMA
token.SEMI
token.PLUS

token.MINUS
token.STAR
token.SLASH
token.VBAR
token.AMPER
token.LESS
token.GREATER
token.EQUAL
token.DOT
token.PERCENT
token.BACKQUOTE
token.LBRACE
token.RBRACE
token.EQEQUAL
token.NOTEQUAL
token.LESSEQUAL
token.GREATEREQUAL
token.TILDE
token.CIRCUMFLEX
token.LEFTSHIFT
token.RIGHTSHIFT
token.DOUBLESTAR
token.PLUSEQUAL
token.MINEQUAL
token.STAREQUAL
token.SLASHEQUAL
token.PERCENTEQUAL
token.AMPEREQUAL
token.VBAREQUAL
token.CIRCUMFLEXEQUAL
token.LEFTSHIFTEQUAL
token.RIGHTSHIFTEQUAL
token.DOUBLESTAREQUAL
token.DOUBLESLASH
token.DOUBLESLASHEQUAL
token.AT
token.OP
token.ERRORTOKEN
token.N_TOKENS
token.NT_OFFSET

See also:

Module parser The second example for the parser module shows how to use the symbol module.

32.6 keyword — Testing for Python keywords

Source code: Lib/keyword.py

This module allows a Python program to determine if a string is a keyword.

keyword.iskeyword(s)
Return true if s is a Python keyword.

keyword.kwlist
: Sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular __future__ statements are in effect, these will be included as well.

32.7 tokenize — Tokenizer for Python source

Source code: Lib/tokenize.py

The tokenize module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing "pretty-printers," including colorizers for on-screen displays.

To simplify token stream handling, all operators and delimiters tokens are returned using the generic token.OP token type. The exact type can be determined by checking the second field (containing the actual token string matched) of the tuple returned from tokenize.generate_tokens() for the character sequence that identifies a specific operator token.

The primary entry point is a generator:

tokenize.generate_tokens(readline)
: The generate_tokens() generator requires one argument, readline, which must be a callable object which provides the same interface as the readline() method of built-in file objects (see section File Objects). Each call to the function should return one line of input as a string. Alternately, readline may be a callable object that signals completion by raising StopIteration.

  The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple (srow, scol) of ints specifying the row and column where the token begins in the source; a 2-tuple (erow, ecol) of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed (the last tuple item) is the logical line; continuation lines are included.

  New in version 2.2.

An older entry point is retained for backward compatibility:

tokenize.tokenize(readline[, tokeneater])
: The tokenize() function accepts two parameters: one representing the input stream, and one providing an output mechanism for tokenize().

  The first parameter, readline, must be a callable object which provides the same interface as the readline() method of built-in file objects (see section File Objects). Each call to the function should return one line of input as a string. Alternately, readline may be a callable object that signals completion by raising StopIteration.

  Changed in version 2.5: Added StopIteration support.

  The second parameter, tokeneater, must also be a callable object. It is called once for each token, with five arguments, corresponding to the tuples generated by generate_tokens().

All constants from the token module are also exported from tokenize, as are two additional token type values that might be passed to the tokeneater function by tokenize():

tokenize.COMMENT
: Token value used to indicate a comment.

tokenize.NL
: Token value used to indicate a non-terminating newline. The NEWLINE token indicates the end of

a logical line of Python code; NL tokens are generated when a logical line of code is continued over multiple physical lines.

Another function is provided to reverse the tokenization process. This is useful for creating tools that tokenize a script, modify the token stream, and write back the modified script.

tokenize.untokenize(iterable)
Converts tokens back into Python source code. The iterable must return sequences with at least two elements, the token type and the token string. Any additional sequence elements are ignored.

The reconstructed script is returned as a single string. The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change.

New in version 2.5.

exception tokenize.TokenError
Raised when either a docstring or expression that may be split over several lines is not completed anywhere in the file, for example:

```
"""Beginning of
docstring
```

or:

```
[1,
 2,
 3
```

Note that unclosed single-quoted strings do not cause an error to be raised. They are tokenized as ERRORTOKEN, followed by the tokenization of their contents.

Example of a script re-writer that transforms float literals into Decimal objects:

```
def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print +21.3e-5*-.1234/81.7'
    >>> decistmt(s)
    "print +Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7')"

    >>> exec(s)
    -3.21716034272e-007
    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7

    """
    result = []
    g = generate_tokens(StringIO(s).readline)   # tokenize the string
    for toknum, tokval, _, _, _  in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
```

(continues on next page)

(continued from previous page)

```
        result.append((toknum, tokval))
    return untokenize(result)
```

32.8 tabnanny — Detection of ambiguous indentation

Source code: Lib/tabnanny.py

For the time being this module is intended to be called as a script. However it is possible to import it into an IDE and use the function check() described below.

Note: The API provided by this module is likely to change in future releases; such changes may not be backward compatible.

---

tabnanny.check(file_or_dir)
: If file_or_dir is a directory and not a symbolic link, then recursively descend the directory tree named by file_or_dir, checking all .py files along the way. If file_or_dir is an ordinary Python source file, it is checked for whitespace related problems. The diagnostic messages are written to standard output using the print statement.

tabnanny.verbose
: Flag indicating whether to print verbose messages. This is incremented by the -v option if called as a script.

tabnanny.filename_only
: Flag indicating whether to print only the filenames of files containing whitespace related problems. This is set to true by the -q option if called as a script.

exception tabnanny.NannyNag
: Raised by process_tokens() if detecting an ambiguous indent. Captured and handled in check().

tabnanny.process_tokens(tokens)
: This function is used by check() to process tokens generated by the tokenize module.

See also:

Module tokenize Lexical scanner for Python source code.

32.9 pyclbr — Python class browser support

Source code: Lib/pyclbr.py

The pyclbr module can be used to determine some limited information about the classes, methods and top-level functions defined in a module. The information provided is sufficient to implement a traditional three-pane class browser. The information is extracted from the source code rather than by importing the module, so this module is safe to use with untrusted code. This restriction makes it impossible to use this module with modules not implemented in Python, including all standard and optional extension modules.

pyclbr.readmodule(module, path=None)
: Read a module and return a dictionary mapping class names to class descriptor objects. The parameter module should be the name of a module as a string; it may be the name of a module within a package. The path parameter should be a sequence, and is used to augment the value of sys.path, which is used to locate module source code.

pyclbr.readmodule_ex(module, path=None)
: Like readmodule(), but the returned dictionary, in addition to mapping class names to class descriptor objects, also maps top-level function names to function descriptor objects. Moreover, if the module being read is a package, the key `'__path__'` in the returned dictionary has as its value a list which contains the package search path.

32.9.1 Class Objects

The Class objects used as values in the dictionary returned by readmodule() and readmodule_ex() provide the following data attributes:

Class.module
: The name of the module defining the class described by the class descriptor.

Class.name
: The name of the class.

Class.super
: A list of Class objects which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by readmodule() are listed as a string with the class name instead of as Class objects.

Class.methods
: A dictionary mapping method names to line numbers.

Class.file
: Name of the file containing the class statement defining the class.

Class.lineno
: The line number of the class statement within the file named by file.

32.9.2 Function Objects

The Function objects used as values in the dictionary returned by readmodule_ex() provide the following attributes:

Function.module
: The name of the module defining the function described by the function descriptor.

Function.name
: The name of the function.

Function.file
: Name of the file containing the def statement defining the function.

Function.lineno
: The line number of the def statement within the file named by file.

32.10 py_compile — Compile Python source files

Source code: Lib/py_compile.py

The py_compile module provides a function to generate a byte-code file from a source file, and another function used when the module source file is invoked as a script.

Though not often needed, this function can be useful when installing modules for shared use, especially if some of the users may not have permission to write the byte-code cache files in the directory containing the source code.

exception py_compile.PyCompileError
: Exception raised when an error occurs while attempting to compile the file.

py_compile.compile(file[, cfile[, dfile[, doraise]]])
: Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file named file. The byte-code is written to cfile, which defaults to file + `'c'` (`'o'` if optimization is enabled in the current interpreter). If dfile is specified, it is used as the name of the source file in error messages instead of file. If doraise is true, a PyCompileError is raised when an error is encountered while compiling file. If doraise is false (the default), an error string is written to sys.stderr, but no exception is raised.

py_compile.main([args])
: Compile several source files. The files named in args (or on the command line, if args is not specified) are compiled and the resulting bytecode is cached in the normal manner. This function does not search a directory structure to locate source files; it only compiles files named explicitly. If `'-'` is the only parameter in args, the list of files is taken from standard input.

  Changed in version 2.7: Added support for `'-'`.

When this module is run as a script, the main() is used to compile all the files named on the command line. The exit status is nonzero if one of the files could not be compiled.

Changed in version 2.6: Added the nonzero exit status when module is run as a script.

See also:

Module compileall Utilities to compile all Python source files in a directory tree.

32.11 compileall — Byte-compile Python libraries

Source code: Lib/compileall.py

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree. This module can be used to create the cached byte-code files at library installation time, which makes them available for use even by users who don't have write permission to the library directories.

32.11.1 Command-line use

This module can work as a script (using python -m compileall) to compile Python sources.

directory ...

file ...
: Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was -l <directories from sys.path>.

-l
: Do not recurse into subdirectories, only compile source code files directly contained in the named or implied directories.

-f
: Force rebuild even if timestamps are up-to-date.

-q
: Do not print the list of files compiled, print only error messages.

-d destdir
: Directory prepended to the path to each file being compiled. This will appear in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

-x regex
: regex is used to search the full path to each file considered for compilation, and if the regex produces a match, the file is skipped.

-i list
: Read the file list and add each line that it contains to the list of files and directories to compile. If list is -, read lines from stdin.

Changed in version 2.7: Added the -i option.

32.11.2 Public functions

compileall.compile_dir(dir[, maxlevels[, ddir[, force[, rx[, quiet]]]]])
: Recursively descend the directory tree named by dir, compiling all .py files along the way.

  The maxlevels parameter is used to limit the depth of the recursion; it defaults to 10.

  If ddir is given, it is prepended to the path to each file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

  If force is true, modules are re-compiled even if the timestamps are up to date.

  If rx is given, its search method is called on the complete path to each file considered for compilation, and if it returns a true value, the file is skipped.

  If quiet is true, nothing is printed to the standard output unless errors occur.

compileall.compile_file(fullname[, ddir[, force[, rx[, quiet]]]])
: Compile the file with path fullname.

  If ddir is given, it is prepended to the path to the file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

  If rx is given, its search method is passed the full path name to the file being compiled, and if it returns a true value, the file is not compiled and True is returned.

  If quiet is true, nothing is printed to the standard output unless errors occur.

  New in version 2.7.

compileall.compile_path([skip_curdir[, maxlevels[, force]]])
Byte-compile all the .py files found along sys.path. If skip_curdir is true (the default), the current directory is not included in the search. All other parameters are passed to the compile_dir() function. Note that unlike the other compile functions, maxlevels defaults to 0.

To force a recompile of all the .py files in the Lib/ subdirectory and all its subdirectories:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)
```

See also:

Module py_compile Byte-compile a single source file.

32.12 dis — Disassembler for Python bytecode

Source code: Lib/dis.py

The dis module supports the analysis of CPython bytecode by disassembling it. The CPython bytecode which this module takes as an input is defined in the file Include/opcode.h and used by the compiler and the interpreter.

CPython implementation detail: Bytecode is an implementation detail of the CPython interpreter! No guarantees are made that bytecode will not be added, removed, or changed between versions of Python. Use of this module should not be considered to work across Python VMs or Python releases.

Example: Given the function myfunc():

```
def myfunc(alist):
    return len(alist)
```

the following command can be used to get the disassembly of myfunc():

```
>>> dis.dis(myfunc)
 2           0 LOAD_GLOBAL              0 (len)
             3 LOAD_FAST                0 (alist)
             6 CALL_FUNCTION            1
             9 RETURN_VALUE
```

(The "2" is a line number).

The dis module defines the following functions and constants:

dis.dis([bytesource])
Disassemble the bytesource object. bytesource can denote either a module, a class, a method, a function, or a code object. For a module, it disassembles all functions. For a class, it disassembles all methods. For a single code sequence, it prints one line per bytecode instruction. If no object is provided, it disassembles the last traceback.

dis.distb([tb])
Disassembles the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.

dis.disassemble(code[, lasti])
Disassembles a code object, indicating the last instruction if lasti was provided. The output is divided in the following columns:

1. the line number, for the first instruction of each line
2. the current instruction, indicated as -->,
3. a labelled instruction, indicated with >>,
4. the address of the instruction,
5. the operation code name,
6. operation parameters, and
7. interpretation of the parameters in parentheses.

The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

dis.disco(code[, lasti])
A synonym for disassemble(). It is more convenient to type, and kept for compatibility with earlier Python releases.

dis.findlinestarts(code)
This generator function uses the co_firstlineno and co_lnotab attributes of the code object code to find the offsets which are starts of lines in the source code. They are generated as (offset, lineno) pairs.

dis.findlabels(code)
Detect all offsets in the code object code which are jump targets, and return a list of these offsets.

dis.opname
Sequence of operation names, indexable using the bytecode.

dis.opmap
Dictionary mapping operation names to bytecodes.

dis.cmp_op
Sequence of all compare operation names.

dis.hasconst
Sequence of bytecodes that access a constant.

dis.hasfree
Sequence of bytecodes that access a free variable.

dis.hasname
Sequence of bytecodes that access an attribute by name.

dis.hasjrel
Sequence of bytecodes that have a relative jump target.

dis.hasjabs
Sequence of bytecodes that have an absolute jump target.

dis.haslocal
Sequence of bytecodes that access a local variable.

dis.hascompare
Sequence of bytecodes of Boolean operations.

32.12.1 Python Bytecode Instructions

The Python compiler currently generates the following bytecode instructions.

STOP_CODE()
: Indicates end-of-code to the compiler, not used by the interpreter.

NOP()
: Do nothing code. Used as a placeholder by the bytecode optimizer.

POP_TOP()
: Removes the top-of-stack (TOS) item.

ROT_TWO()
: Swaps the two top-most stack items.

ROT_THREE()
: Lifts second and third stack item one position up, moves top down to position three.

ROT_FOUR()
: Lifts second, third and forth stack item one position up, moves top down to position four.

DUP_TOP()
: Duplicates the reference on top of the stack.

Unary Operations take the top of the stack, apply the operation, and push the result back on the stack.

UNARY_POSITIVE()
: Implements TOS = +TOS.

UNARY_NEGATIVE()
: Implements TOS = -TOS.

UNARY_NOT()
: Implements TOS = not TOS.

UNARY_CONVERT()
: Implements TOS = `TOS`.

UNARY_INVERT()
: Implements TOS = ~TOS.

GET_ITER()
: Implements TOS = iter(TOS).

Binary operations remove the top of the stack (TOS) and the second top-most stack item (TOS1) from the stack. They perform the operation, and put the result back on the stack.

BINARY_POWER()
: Implements TOS = TOS1 ** TOS.

BINARY_MULTIPLY()
: Implements TOS = TOS1 * TOS.

BINARY_DIVIDE()
: Implements TOS = TOS1 / TOS when from __future__ import division is not in effect.

BINARY_FLOOR_DIVIDE()
: Implements TOS = TOS1 // TOS.

BINARY_TRUE_DIVIDE()
: Implements TOS = TOS1 / TOS when from __future__ import division is in effect.

BINARY_MODULO()
Implements TOS = TOS1 % TOS.

BINARY_ADD()
Implements TOS = TOS1 + TOS.

BINARY_SUBTRACT()
Implements TOS = TOS1 - TOS.

BINARY_SUBSCR()
Implements TOS = TOS1[TOS].

BINARY_LSHIFT()
Implements TOS = TOS1 << TOS.

BINARY_RSHIFT()
Implements TOS = TOS1 >> TOS.

BINARY_AND()
Implements TOS = TOS1 & TOS.

BINARY_XOR()
Implements TOS = TOS1 ^ TOS.

BINARY_OR()
Implements TOS = TOS1 | TOS.

In-place operations are like binary operations, in that they remove TOS and TOS1, and push the result back on the stack, but the operation is done in-place when TOS1 supports it, and the resulting TOS may be (but does not have to be) the original TOS1.

INPLACE_POWER()
Implements in-place TOS = TOS1 ** TOS.

INPLACE_MULTIPLY()
Implements in-place TOS = TOS1 * TOS.

INPLACE_DIVIDE()
Implements in-place TOS = TOS1 / TOS when from __future__ import division is not in effect.

INPLACE_FLOOR_DIVIDE()
Implements in-place TOS = TOS1 // TOS.

INPLACE_TRUE_DIVIDE()
Implements in-place TOS = TOS1 / TOS when from __future__ import division is in effect.

INPLACE_MODULO()
Implements in-place TOS = TOS1 % TOS.

INPLACE_ADD()
Implements in-place TOS = TOS1 + TOS.

INPLACE_SUBTRACT()
Implements in-place TOS = TOS1 - TOS.

INPLACE_LSHIFT()
Implements in-place TOS = TOS1 << TOS.

INPLACE_RSHIFT()
Implements in-place TOS = TOS1 >> TOS.

INPLACE_AND()
Implements in-place TOS = TOS1 & TOS.

INPLACE_XOR()
: Implements in-place TOS = TOS1 ^ TOS.

INPLACE_OR()
: Implements in-place TOS = TOS1 | TOS.

The slice opcodes take up to three parameters.

SLICE+0()
: Implements TOS = TOS[:].

SLICE+1()
: Implements TOS = TOS1[TOS:].

SLICE+2()
: Implements TOS = TOS1[:TOS].

SLICE+3()
: Implements TOS = TOS2[TOS1:TOS].

Slice assignment needs even an additional parameter. As any statement, they put nothing on the stack.

STORE_SLICE+0()
: Implements TOS[:] = TOS1.

STORE_SLICE+1()
: Implements TOS1[TOS:] = TOS2.

STORE_SLICE+2()
: Implements TOS1[:TOS] = TOS2.

STORE_SLICE+3()
: Implements TOS2[TOS1:TOS] = TOS3.

DELETE_SLICE+0()
: Implements del TOS[:].

DELETE_SLICE+1()
: Implements del TOS1[TOS:].

DELETE_SLICE+2()
: Implements del TOS1[:TOS].

DELETE_SLICE+3()
: Implements del TOS2[TOS1:TOS].

STORE_SUBSCR()
: Implements TOS1[TOS] = TOS2.

DELETE_SUBSCR()
: Implements del TOS1[TOS].

Miscellaneous opcodes.

PRINT_EXPR()
: Implements the expression statement for the interactive mode. TOS is removed from the stack and printed. In non-interactive mode, an expression statement is terminated with POP_TOP.

PRINT_ITEM()
: Prints TOS to the file-like object bound to sys.stdout. There is one such instruction for each item in the print statement.

PRINT_ITEM_TO()
: Like PRINT_ITEM, but prints the item second from TOS to the file-like object at TOS. This is used by the extended print statement.

PRINT_NEWLINE()
: Prints a new line on sys.stdout. This is generated as the last operation of a print statement, unless the statement ends with a comma.

PRINT_NEWLINE_TO()
: Like PRINT_NEWLINE, but prints the new line on the file-like object on the TOS. This is used by the extended print statement.

BREAK_LOOP()
: Terminates a loop due to a break statement.

CONTINUE_LOOP(target)
: Continues a loop due to a continue statement. target is the address to jump to (which should be a FOR_ITER instruction).

LIST_APPEND(i)
: Calls list.append(TOS[-i], TOS). Used to implement list comprehensions. While the appended value is popped off, the list object remains on the stack so that it is available for further iterations of the loop.

LOAD_LOCALS()
: Pushes a reference to the locals of the current scope on the stack. This is used in the code for a class definition: After the class body is evaluated, the locals are passed to the class definition.

RETURN_VALUE()
: Returns with TOS to the caller of the function.

YIELD_VALUE()
: Pops TOS and yields it from a generator.

IMPORT_STAR()
: Loads all symbols not starting with `'_'` directly from the module TOS to the local namespace. The module is popped after loading all names. This opcode implements from module import *.

EXEC_STMT()
: Implements exec TOS2,TOS1,TOS. The compiler fills missing optional parameters with None.

POP_BLOCK()
: Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

END_FINALLY()
: Terminates a finally clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

BUILD_CLASS()
: Creates a new class object. TOS is the methods dictionary, TOS1 the tuple of the names of the base classes, and TOS2 the class name.

SETUP_WITH(delta)
: This opcode performs several operations before a with block starts. First, it loads __exit__() from the context manager and pushes it onto the stack for later use by WITH_CLEANUP. Then, __enter__() is called, and a finally block pointing to delta is pushed. Finally, the result of calling the enter method is pushed onto the stack. The next opcode will either ignore it (POP_TOP), or store it in (a) variable(s) (STORE_FAST, STORE_NAME, or UNPACK_SEQUENCE).

WITH_CLEANUP()
: Cleans up the stack when a with statement block exits. On top of the stack are 1–3 values indicating

how/why the finally clause was entered:

- TOP = None
- (TOP, SECOND) = (WHY_{RETURN,CONTINUE}), retval
- TOP = WHY_*; no retval below it
- (TOP, SECOND, THIRD) = exc_info()

Under them is EXIT, the context manager's __exit__() bound method.

In the last case, EXIT(TOP, SECOND, THIRD) is called, otherwise EXIT(None, None, None).

EXIT is removed from the stack, leaving the values above it in the same order. In addition, if the stack represents an exception, and the function call returns a 'true' value, this information is "zapped", to prevent END_FINALLY from re-raising the exception. (But non-local gotos should still be resumed.)

All of the following opcodes expect arguments. An argument is two bytes, with the more significant byte last.

STORE_NAME(namei)
Implements name = TOS. namei is the index of name in the attribute co_names of the code object. The compiler tries to use STORE_FAST or STORE_GLOBAL if possible.

DELETE_NAME(namei)
Implements del name, where namei is the index into co_names attribute of the code object.

UNPACK_SEQUENCE(count)
Unpacks TOS into count individual values, which are put onto the stack right-to-left.

DUP_TOPX(count)
Duplicate count items, keeping them in the same order. Due to implementation limits, count should be between 1 and 5 inclusive.

STORE_ATTR(namei)
Implements TOS.name = TOS1, where namei is the index of name in co_names.

DELETE_ATTR(namei)
Implements del TOS.name, using namei as index into co_names.

STORE_GLOBAL(namei)
Works as STORE_NAME, but stores the name as a global.

DELETE_GLOBAL(namei)
Works as DELETE_NAME, but deletes a global name.

LOAD_CONST(consti)
Pushes co_consts[consti] onto the stack.

LOAD_NAME(namei)
Pushes the value associated with co_names[namei] onto the stack.

BUILD_TUPLE(count)
Creates a tuple consuming count items from the stack, and pushes the resulting tuple onto the stack.

BUILD_LIST(count)
Works as BUILD_TUPLE, but creates a list.

BUILD_SET(count)
Works as BUILD_TUPLE, but creates a set.

New in version 2.7.

BUILD_MAP(count)
: Pushes a new dictionary object onto the stack. The dictionary is pre-sized to hold count entries.

LOAD_ATTR(namei)
: Replaces TOS with getattr(TOS, co_names[namei]).

COMPARE_OP(opname)
: Performs a Boolean operation. The operation name can be found in cmp_op[opname].

IMPORT_NAME(namei)
: Imports the module co_names[namei]. TOS and TOS1 are popped and provide the fromlist and level arguments of __import__(). The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent STORE_FAST instruction modifies the namespace.

IMPORT_FROM(namei)
: Loads the attribute co_names[namei] from the module found in TOS. The resulting object is pushed onto the stack, to be subsequently stored by a STORE_FAST instruction.

JUMP_FORWARD(delta)
: Increments bytecode counter by delta.

POP_JUMP_IF_TRUE(target)
: If TOS is true, sets the bytecode counter to target. TOS is popped.

POP_JUMP_IF_FALSE(target)
: If TOS is false, sets the bytecode counter to target. TOS is popped.

JUMP_IF_TRUE_OR_POP(target)
: If TOS is true, sets the bytecode counter to target and leaves TOS on the stack. Otherwise (TOS is false), TOS is popped.

JUMP_IF_FALSE_OR_POP(target)
: If TOS is false, sets the bytecode counter to target and leaves TOS on the stack. Otherwise (TOS is true), TOS is popped.

JUMP_ABSOLUTE(target)
: Set bytecode counter to target.

FOR_ITER(delta)
: TOS is an iterator. Call its next() method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted TOS is popped, and the bytecode counter is incremented by delta.

LOAD_GLOBAL(namei)
: Loads the global named co_names[namei] onto the stack.

SETUP_LOOP(delta)
: Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of delta bytes.

SETUP_EXCEPT(delta)
: Pushes a try block from a try-except clause onto the block stack. delta points to the first except block.

SETUP_FINALLY(delta)
: Pushes a try block from a try-except clause onto the block stack. delta points to the finally block.

STORE_MAP()
: Store a key and value pair in a dictionary. Pops the key and value while leaving the dictionary on the stack.

LOAD_FAST(var_num)
Pushes a reference to the local co_varnames[var_num] onto the stack.

STORE_FAST(var_num)
Stores TOS into the local co_varnames[var_num].

DELETE_FAST(var_num)
Deletes local co_varnames[var_num].

LOAD_CLOSURE(i)
Pushes a reference to the cell contained in slot i of the cell and free variable storage. The name of the variable is co_cellvars[i] if i is less than the length of co_cellvars. Otherwise it is co_freevars[i - len(co_cellvars)].

LOAD_DEREF(i)
Loads the cell contained in slot i of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

STORE_DEREF(i)
Stores TOS into the cell contained in slot i of the cell and free variable storage.

SET_LINENO(lineno)
This opcode is obsolete.

RAISE_VARARGS(argc)
Raises an exception. argc indicates the number of arguments to the raise statement, ranging from 0 to 3. The handler will find the traceback as TOS2, the parameter as TOS1, and the exception as TOS.

CALL_FUNCTION(argc)
Calls a callable object. The low byte of argc indicates the number of positional arguments, the high byte the number of keyword arguments. The stack contains keyword arguments on top (if any), then the positional arguments below that (if any), then the callable object to call below that. Each keyword argument is represented with two values on the stack: the argument's name, and its value, with the argument's value above the name on the stack. The positional arguments are pushed in the order that they are passed in to the callable object, with the right-most positional argument on top. CALL_FUNCTION pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

MAKE_FUNCTION(argc)
Pushes a new function object on the stack. TOS is the code associated with the function. The function object is defined to have argc default parameters, which are found below TOS.

MAKE_CLOSURE(argc)
Creates a new function object, sets its func_closure slot, and pushes it on the stack. TOS is the code associated with the function, TOS1 the tuple containing cells for the closure's free variables. The function also has argc default parameters, which are found below the cells.

BUILD_SLICE(argc)
Pushes a slice object on the stack. argc must be 2 or 3. If it is 2, slice(TOS1, TOS) is pushed; if it is 3, slice(TOS2, TOS1, TOS) is pushed. See the slice() built-in function for more information.

EXTENDED_ARG(ext)
Prefixes any opcode which has an argument too big to fit into the default two bytes. ext holds two additional bytes which, taken together with the subsequent opcode's argument, comprise a four-byte argument, ext being the two most-significant bytes.

CALL_FUNCTION_VAR(argc)
Calls a callable object, similarly to CALL_FUNCTION. argc represents the number of keyword and positional arguments, identically to CALL_FUNCTION. The top of the stack contains an iterable object containing additional positional arguments. Below that are keyword arguments (if any), positional

arguments (if any) and a callable object, identically to CALL_FUNCTION. Before the callable object is called, the iterable object is "unpacked" and its contents are appended to the positional arguments passed in. The iterable object is ignored when computing the value of argc.

CALL_FUNCTION_KW(argc)
Calls a callable object, similarly to CALL_FUNCTION. argc represents the number of keyword and positional arguments, identically to CALL_FUNCTION. The top of the stack contains a mapping object containing additional keyword arguments. Below that are keyword arguments (if any), positional arguments (if any) and a callable object, identically to CALL_FUNCTION. Before the callable is called, the mapping object at the top of the stack is "unpacked" and its contents are appended to the keyword arguments passed in. The mapping object at the top of the stack is ignored when computing the value of argc.

CALL_FUNCTION_VAR_KW(argc)
Calls a callable object, similarly to CALL_FUNCTION_VAR and CALL_FUNCTION_KW. argc represents the number of keyword and positional arguments, identically to CALL_FUNCTION. The top of the stack contains a mapping object, as per CALL_FUNCTION_KW. Below that is an iterable object, as per CALL_FUNCTION_VAR. Below that are keyword arguments (if any), positional arguments (if any) and a callable object, identically to CALL_FUNCTION. Before the callable is called, the mapping object and iterable object are each "unpacked" and their contents passed in as keyword and positional arguments respectively, identically to CALL_FUNCTION_VAR and CALL_FUNCTION_KW. The mapping object and iterable object are both ignored when computing the value of argc.

HAVE_ARGUMENT()
This is not really an opcode. It identifies the dividing line between opcodes which don't take arguments < HAVE_ARGUMENT and those which do >= HAVE_ARGUMENT.

32.13 pickletools — Tools for pickle developers

New in version 2.3.

Source code: Lib/pickletools.py

This module contains various constants relating to the intimate details of the pickle module, some lengthy comments about the implementation, and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers who are working on the pickle and cPickle implementations; ordinary users of the pickle module probably won't find the pickletools module relevant.

pickletools.dis(pickle, out=None, memo=None, indentlevel=4)
Outputs a symbolic disassembly of the pickle to the file-like object out, defaulting to sys.stdout. pickle can be a string or a file-like object. memo can be a Python dictionary that will be used as the pickle's memo; it can be used to perform disassemblies across multiple pickles created by the same pickler. Successive levels, indicated by MARK opcodes in the stream, are indented by indentlevel spaces.

pickletools.genops(pickle)
Provides an iterator over all of the opcodes in a pickle, returning a sequence of (opcode, arg, pos) triples. opcode is an instance of an OpcodeInfo class; arg is the decoded value, as a Python object, of the opcode's argument; pos is the position at which this opcode is located. pickle can be a string or a file-like object.

pickletools.optimize(picklestring)
Returns a new equivalent pickle string after eliminating unused PUT opcodes. The optimized pickle is shorter, takes less transmission time, requires less storage space, and unpickles more efficiently.

New in version 2.6.

CHAPTER

THIRTYTHREE

PYTHON COMPILER PACKAGE

Deprecated since version 2.6: The compiler package has been removed in Python 3.

The Python compiler package is a tool for analyzing Python source code and generating Python bytecode. The compiler contains libraries to generate an abstract syntax tree from Python source code and to generate Python bytecode from the tree.

The compiler package is a Python source to bytecode translator written in Python. It uses the built-in parser and standard parser module to generate a concrete syntax tree. This tree is used to generate an abstract syntax tree (AST) and then Python bytecode.

The full functionality of the package duplicates the built-in compiler provided with the Python interpreter. It is intended to match its behavior almost exactly. Why implement another compiler that does the same thing? The package is useful for a variety of purposes. It can be modified more easily than the built-in compiler. The AST it generates is useful for analyzing Python source code.

This chapter explains how the various components of the compiler package work. It blends reference material with a tutorial.

33.1 The basic interface

The top-level of the package defines four functions. If you import compiler, you will get these functions and a collection of modules contained in the package.

compiler.parse(buf)
: Returns an abstract syntax tree for the Python source code in buf. The function raises SyntaxError if there is an error in the source code. The return value is a compiler.ast.Module instance that contains the tree.

compiler.parseFile(path)
: Return an abstract syntax tree for the Python source code in the file specified by path. It is equivalent to parse(open(path).read()).

compiler.walk(ast, visitor[, verbose])
: Do a pre-order walk over the abstract syntax tree ast. Call the appropriate method on the visitor instance for each node encountered.

compiler.compile(source, filename, mode, flags=None, dont_inherit=None)
: Compile the string source, a Python module, statement or expression, into a code object that can be executed by the exec statement or eval(). This function is a replacement for the built-in compile() function.

  The filename will be used for run-time error messages.

The mode must be 'exec' to compile a module, 'single' to compile a single (interactive) statement, or 'eval' to compile an expression.

The flags and dont_inherit arguments affect future-related statements, but are not supported yet.

compiler.compileFile(source)
Compiles the file source and generates a .pyc file.

The compiler package contains the following modules: ast, consts, future, misc, pyassem, pycodegen, symbols, transformer, and visitor.

33.2 Limitations

There are some problems with the error checking of the compiler package. The interpreter detects syntax errors in two distinct phases. One set of errors is detected by the interpreter's parser, the other set by the compiler. The compiler package relies on the interpreter's parser, so it get the first phases of error checking for free. It implements the second phase itself, and that implementation is incomplete. For example, the compiler package does not raise an error if a name appears more than once in an argument list: def f(x, x): ...

A future version of the compiler should fix these problems.

33.3 Python Abstract Syntax

The compiler.ast module defines an abstract syntax for Python. In the abstract syntax tree, each node represents a syntactic construct. The root of the tree is Module object.

The abstract syntax offers a higher level interface to parsed Python source code. The parser module and the compiler written in C for the Python interpreter use a concrete syntax tree. The concrete syntax is tied closely to the grammar description used for the Python parser. Instead of a single node for a construct, there are often several levels of nested nodes that are introduced by Python's precedence rules.

The abstract syntax tree is created by the compiler.transformer module. The transformer relies on the built-in Python parser to generate a concrete syntax tree. It generates an abstract syntax tree from the concrete tree.

The transformer module was created by Greg Stein and Bill Tutt for an experimental Python-to-C compiler. The current version contains a number of modifications and improvements, but the basic form of the abstract syntax and of the transformer are due to Stein and Tutt.

33.3.1 AST Nodes

The compiler.ast module is generated from a text file that describes each node type and its elements. Each node type is represented as a class that inherits from the abstract base class compiler.ast.Node and defines a set of named attributes for child nodes.

class compiler.ast.Node
The Node instances are created automatically by the parser generator. The recommended interface for specific Node instances is to use the public attributes to access child nodes. A public attribute may be bound to a single node or to a sequence of nodes, depending on the Node type. For example, the bases attribute of the Class node, is bound to a list of base class nodes, and the doc attribute is bound to a single node.

Each Node instance has a lineno attribute which may be None. XXX Not sure what the rules are for which nodes will have a useful lineno.

All Node objects offer the following methods:

getChildren()
: Returns a flattened list of the child nodes and objects in the order they occur. Specifically, the order of the nodes is the order in which they appear in the Python grammar. Not all of the children are Node instances. The names of functions and classes, for example, are plain strings.

getChildNodes()
: Returns a flattened list of the child nodes in the order they occur. This method is like getChildren(), except that it only returns those children that are Node instances.

Two examples illustrate the general structure of Node classes. The while statement is defined by the following grammar production:

```
while_stmt:     "while" expression ":" suite
                ["else" ":" suite]
```

The While node has three attributes: test, body, and else_. (If the natural name for an attribute is also a Python reserved word, it can't be used as an attribute name. An underscore is appended to the word to make it a legal identifier, hence else_ instead of else.)

The if statement is more complicated because it can include several tests.

```
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
```

The If node only defines two attributes: tests and else_. The tests attribute is a sequence of test expression, consequent body pairs. There is one pair for each if/elif clause. The first element of the pair is the test expression. The second elements is a Stmt node that contains the code to execute if the test is true.

The getChildren() method of If returns a flat list of child nodes. If there are three if/elif clauses and no else clause, then getChildren() will return a list of six elements: the first test expression, the first Stmt, the second text expression, etc.

The following table lists each of the Node subclasses defined in compiler.ast and each of the public attributes available on their instances. The values of most of the attributes are themselves Node instances or sequences of instances. When the value is something other than an instance, the type is noted in the comment. The attributes are listed in the order in which they are returned by getChildren() and getChildNodes().

Node type	Attribute	Value
Add	left	left operand
	right	right operand
And	nodes	list of operands
AssAttr		attribute as target of assignment
	expr	expression on the left-hand side of the dot
	attrname	the attribute name, a string
	flags	XXX
AssList	nodes	list of list elements being assigned to
AssName	name	name being assigned to
	flags	XXX
AssTuple	nodes	list of tuple elements being assigned to
Assert	test	the expression to be tested
	fail	the value of the AssertionError
Assign	nodes	a list of assignment targets, one per equal sign

Continued on next page

Table 1 – continued from previous page

Node type	Attribute	Value
	expr	the value being assigned
AugAssign	node	
	op	
	expr	
Backquote	expr	
Bitand	nodes	
Bitor	nodes	
Bitxor	nodes	
Break		
CallFunc	node	expression for the callee
	args	a list of arguments
	star_args	the extended *-arg value
	dstar_args	the extended **-arg value
Class	name	the name of the class, a string
	bases	a list of base classes
	doc	doc string, a string or None
	code	the body of the class statement
Compare	expr	
	ops	
Const	value	
Continue		
Decorators	nodes	List of function decorator expressions
Dict	items	
Discard	expr	
Div	left	
	right	
Ellipsis		
Expression	node	
Exec	expr	
	locals	
	globals	
FloorDiv	left	
	right	
For	assign	
	list	
	body	
	else_	
From	modname	
	names	
Function	decorators	Decorators or None
	name	name used in def, a string
	argnames	list of argument names, as strings
	defaults	list of default values
	flags	xxx
	doc	doc string, a string or None
	code	the body of the function
GenExpr	code	
GenExprFor	assign	
	iter	

Continued on next page

Table 1 – continued from previous page

Node type	Attribute	Value
	ifs	
GenExprIf	test	
GenExprInner	expr	
	quals	
Getattr	expr	
	attrname	
Global	names	
If	tests	
	else_	
Import	names	
Invert	expr	
Keyword	name	
	expr	
Lambda	argnames	
	defaults	
	flags	
	code	
LeftShift	left	
	right	
List	nodes	
ListComp	expr	
	quals	
ListCompFor	assign	
	list	
	ifs	
ListCompIf	test	
Mod	left	
	right	
Module	doc	doc string, a string or None
	node	body of the module, a Stmt
Mul	left	
	right	
Name	name	
Not	expr	
Or	nodes	
Pass		
Power	left	
	right	
Print	nodes	
	dest	
Printnl	nodes	
	dest	
Raise	expr1	
	expr2	
	expr3	
Return	value	
RightShift	left	
	right	
Slice	expr	

Continued on next page

Table 1 – continued from previous page

Node type	Attribute	Value
	flags	
	lower	
	upper	
Sliceobj	nodes	list of statements
Stmt	nodes	
Sub	left	
	right	
Subscript	expr	
	flags	
	subs	
TryExcept	body	
	handlers	
	else_	
TryFinally	body	
	final	
Tuple	nodes	
UnaryAdd	expr	
UnarySub	expr	
While	test	
	body	
	else_	
With	expr	
	vars	
	body	
Yield	value	

33.3.2 Assignment nodes

There is a collection of nodes used to represent assignments. Each assignment statement in the source code becomes a single Assign node in the AST. The nodes attribute is a list that contains a node for each assignment target. This is necessary because assignment can be chained, e.g. a = b = 2. Each Node in the list will be one of the following classes: AssAttr, AssList, AssName, or AssTuple.

Each target assignment node will describe the kind of object being assigned to: AssName for a simple name, e.g. a = 1. AssAttr for an attribute assigned, e.g. a.x = 1. AssList and AssTuple for list and tuple expansion respectively, e.g. a, b, c = a_tuple.

The target assignment nodes also have a flags attribute that indicates whether the node is being used for assignment or in a delete statement. The AssName is also used to represent a delete statement, e.g. del x.

When an expression contains several attribute references, an assignment or delete statement will contain only one AssAttr node – for the final attribute reference. The other attribute references will be represented as Getattr nodes in the expr attribute of the AssAttr instance.

33.3.3 Examples

This section shows several simple examples of ASTs for Python source code. The examples demonstrate how to use the parse() function, what the repr of an AST looks like, and how to access attributes of an AST node.

The first module defines a single function. Assume it is stored in doublelib.py.

```
"""This is an example module.

This is the docstring.
"""

def double(x):
    "Return twice the argument"
    return x * 2
```

In the interactive interpreter session below, I have reformatted the long AST reprs for readability. The AST reprs use unqualified class names. If you want to create an instance from a repr, you must import the class names from the compiler.ast module.

```
>>> import compiler
>>> mod = compiler.parseFile("doublelib.py")
>>> mod
Module('This is an example module.\n\nThis is the docstring.\n',
       Stmt([Function(None, 'double', ['x'], [], 0,
                      'Return twice the argument',
                      Stmt([Return(Mul((Name('x'), Const(2))))]))]))
>>> from compiler.ast import *
>>> Module('This is an example module.\n\nThis is the docstring.\n',
...     Stmt([Function(None, 'double', ['x'], [], 0,
...                    'Return twice the argument',
...                    Stmt([Return(Mul((Name('x'), Const(2))))]))]))
Module('This is an example module.\n\nThis is the docstring.\n',
       Stmt([Function(None, 'double', ['x'], [], 0,
                      'Return twice the argument',
                      Stmt([Return(Mul((Name('x'), Const(2))))]))]))
>>> mod.doc
'This is an example module.\n\nThis is the docstring.\n'
>>> for node in mod.node.nodes:
...     print node
...
Function(None, 'double', ['x'], [], 0, 'Return twice the argument',
         Stmt([Return(Mul((Name('x'), Const(2))))]))
>>> func = mod.node.nodes[0]
>>> func.code
Stmt([Return(Mul((Name('x'), Const(2))))])
```

33.4 Using Visitors to Walk ASTs

The visitor pattern is ... The compiler package uses a variant on the visitor pattern that takes advantage of Python's introspection features to eliminate the need for much of the visitor's infrastructure.

The classes being visited do not need to be programmed to accept visitors. The visitor need only define visit methods for classes it is specifically interested in; a default visit method can handle the rest.

XXX The magic visit() method for visitors.

compiler.visitor.walk(tree, visitor[, verbose])

class compiler.visitor.ASTVisitor
: The ASTVisitor is responsible for walking over the tree in the correct order. A walk begins with a call to preorder(). For each node, it checks the visitor argument to preorder() for a method named

'visitNodeType,' where NodeType is the name of the node's class, e.g. for a While node a visitWhile() would be called. If the method exists, it is called with the node as its first argument.

The visitor method for a particular node type can control how child nodes are visited during the walk. The ASTVisitor modifies the visitor argument by adding a visit method to the visitor; this method can be used to visit a particular child node. If no visitor is found for a particular node type, the default() method is called.

ASTVisitor objects have the following methods:

XXX describe extra arguments

default(node[, ...])

dispatch(node[, ...])

preorder(tree, visitor)

33.5 Bytecode Generation

The code generator is a visitor that emits bytecodes. Each visit method can call the emit() method to emit a new bytecode. The basic code generator is specialized for modules, classes, and functions. An assembler converts that emitted instructions to the low-level bytecode format. It handles things like generation of constant lists of code objects and calculation of jump offsets.

CHAPTER

THIRTYFOUR

MISCELLANEOUS SERVICES

The modules described in this chapter provide miscellaneous services that are available in all Python versions. Here's an overview:

34.1 formatter — Generic output formatting

This module supports two interface definitions, each with multiple implementations. The formatter interface is used by the HTMLParser class of the htmllib module, and the writer interface is required by the formatter interface.

Formatter objects transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored; writers need not be able to handle relative changes nor any sort of "change back" operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.

34.1.1 The Formatter Interface

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level:

formatter.AS_IS
: Value which can be used in the font specification passed to the push_font() method described below, or as the new value to any other push_property() method. Pushing the AS_IS value allows the corresponding pop_property() method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects:

formatter.writer
: The writer instance with which the formatter interacts.

formatter.end_paragraph(blanklines)
: Close any open paragraphs and insert at least blanklines before the next paragraph.

formatter.add_line_break()
Add a hard line break if one does not already exist. This does not break the logical paragraph.

formatter.add_hor_rule(*args, **kw)
Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph, but the logical paragraph is not broken. The arguments and keywords are passed on to the writer's send_line_break() method.

formatter.add_flowing_data(data)
Provide data which should be formatted with collapsed whitespace. Whitespace from preceding and successive calls to add_flowing_data() is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

formatter.add_literal_data(data)
Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of data.

formatter.add_label_data(format, counter)
Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the format value is a string, it is interpreted as a format specification for counter, which should be an integer. The result of this formatting becomes the value of the label; if format is not a string it is used as the label value directly. The label value is passed as the only argument to the writer's send_label_data() method. Interpretation of non-string label values is dependent on the associated writer.

Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character `'1'` represents the counter value formatter as an Arabic number, the characters `'A'` and `'a'` represent alphabetic representations of the counter value in upper and lower case, respectively, and `'I'` and `'i'` represent the counter value in Roman numerals, in upper and lower case. Note that the alphabetic and roman transforms require that the counter value be greater than zero.

formatter.flush_softspace()
Send any pending whitespace buffered from a previous call to add_flowing_data() to the associated writer object. This should be called before any direct manipulation of the writer object.

formatter.push_alignment(align)
Push a new alignment setting onto the alignment stack. This may be AS_IS if no change is desired. If the alignment value is changed from the previous setting, the writer's new_alignment() method is called with the align value.

formatter.pop_alignment()
Restore the previous alignment.

formatter.push_font((size, italic, bold, teletype))
Change some or all font properties of the writer object. Properties which are not set to AS_IS are set to the values passed in while others are maintained at their current settings. The writer's new_font() method is called with the fully resolved font specification.

formatter.pop_font()
Restore the previous font.

formatter.push_margin(margin)
Increase the number of left margin indentations by one, associating the logical tag margin with the new indentation. The initial margin level is 0. Changed values of the logical tag must be true values; false values other than AS_IS are not sufficient to change the margin.

formatter.pop_margin()
Restore the previous margin.

formatter.push_style(*styles)
Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple representing the entire stack, including AS_IS values, is passed to the writer's new_styles() method.

formatter.pop_style([n=1])
Pop the last n style specifications passed to push_style(). A tuple representing the revised stack, including AS_IS values, is passed to the writer's new_styles() method.

formatter.set_spacing(spacing)
Set the spacing style for the writer.

formatter.assert_line_data([flag=1])
Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when the writer has been manipulated directly. The optional flag argument can be set to false if the writer manipulations produced a hard line break at the end of the output.

34.1.2 Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

class formatter.NullFormatter([writer])
A formatter which does nothing. If writer is omitted, a NullWriter instance is created. No methods of the writer are called by NullFormatter instances. Implementations should inherit from this class if implementing a writer interface but don't need to inherit any implementation.

class formatter.AbstractFormatter(writer)
The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be used directly in most circumstances. It has been used to implement a full-featured World Wide Web browser.

34.1.3 The Writer Interface

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below are the required interfaces which all writers must support once initialized. Note that while most applications can use the AbstractFormatter class as a formatter, the writer must typically be provided by the application.

writer.flush()
Flush any buffered output or device control events.

writer.new_alignment(align)
Set the alignment style. The align value can be any object, but by convention is a string or None, where None indicates that the writer's "preferred" alignment should be used. Conventional align values are `'left'`, `'center'`, `'right'`, and `'justify'`.

writer.new_font(font)
Set the font style. The value of font will be None, indicating that the device's default font should be used, or a tuple of the form (size, italic, bold, teletype). Size will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The italic, bold, and teletype values are Boolean values specifying which of those font attributes should be used.

writer.new_margin(margin, level)
Set the margin level to the integer level and the logical tag to margin. Interpretation of the logical tag is at the writer's discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of level.

writer.new_spacing(spacing)
Set the spacing style to spacing.

writer.new_styles(styles)
Set additional styles. The styles value is a tuple of arbitrary values; the value AS_IS should be ignored. The styles tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

writer.send_line_break()
Break the current line.

writer.send_paragraph(blankline)
Produce a paragraph separation of at least blankline blank lines, or the equivalent. The blankline value will be an integer. Note that the implementation will receive a call to send_line_break() before this call if a line break is needed; this method should not include ending the last line of the paragraph. It is only responsible for vertical spacing between paragraphs.

writer.send_hor_rule(*args, **kw)
Display a horizontal rule on the output device. The arguments to this method are entirely application- and writer-specific, and should be interpreted with care. The method implementation may assume that a line break has already been issued via send_line_break().

writer.send_flowing_data(data)
Output character data which may be word-wrapped and re-flowed as needed. Within any sequence of calls to this method, the writer may assume that spans of multiple whitespace characters have been collapsed to single space characters.

writer.send_literal_data(data)
Output character data which has already been formatted for display. Generally, this should be interpreted to mean that line breaks indicated by newline characters should be preserved and no new line breaks should be introduced. The data may contain embedded newline and tab characters, unlike data provided to the send_formatted_data() interface.

writer.send_label_data(data)
Set data to the left of the current left margin, if possible. The value of data is not restricted; treatment of non-string values is entirely application- and writer-dependent. This method will only be called at the beginning of a line.

34.1.4 Writer Implementations

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the NullWriter class.

class formatter.NullWriter
A writer which only provides the interface definition; no actions are taken on any methods. This should be the base class for all writers which do not need to inherit any implementation methods.

class formatter.AbstractWriter
A writer which can be used in debugging formatters, but not much else. Each method simply announces itself by printing its name and arguments on standard output.

class formatter.DumbWriter(file=None, maxcol=72)
Simple writer class which writes output on the file object passed in as file or, if file is None, on standard

output. The output is simply word-wrapped to the number of columns specified by maxcol. This class is suitable for reflowing a sequence of paragraphs.

CHAPTER

THIRTYFIVE

MS WINDOWS SPECIFIC SERVICES

This chapter describes modules that are only available on MS Windows platforms.

35.1 msilib — Read and write Microsoft Installer files

New in version 2.5.

The msilib supports the creation of Microsoft Installer (.msi) files. Because these files often contain an embedded "cabinet" file (.cab), it also exposes an API to create CAB files. Support for reading .cab files is currently not implemented; read support for the .msi database is possible.

This package aims to provide complete access to all tables in an .msi file, therefore, it is a fairly low-level API. Two primary applications of this package are the distutils command bdist_msi, and the creation of Python installer package itself (although that currently uses a different version of msilib).

The package contents can be roughly split into four parts: low-level CAB routines, low-level MSI routines, higher-level MSI routines, and standard table structures.

msilib.FCICreate(cabname, files)
: Create a new CAB file named cabname. files must be a list of tuples, each containing the name of the file on disk, and the name of the file inside the CAB file.

  The files are added to the CAB file in the order they appear in the list. All files are added into a single CAB file, using the MSZIP compression algorithm.

  Callbacks to Python for the various steps of MSI creation are currently not exposed.

msilib.UuidCreate()
: Return the string representation of a new unique identifier. This wraps the Windows API functions UuidCreate() and UuidToString().

msilib.OpenDatabase(path, persist)
: Return a new database object by calling MsiOpenDatabase. path is the file name of the MSI file; persist can be one of the constants MSIDBOPEN_CREATEDIRECT, MSIDBOPEN_CREATE, MSIDBOPEN_DIRECT, MSIDBOPEN_READONLY, or MSIDBOPEN_TRANSACT, and may include the flag MSIDBOPEN_PATCHFILE. See the Microsoft documentation for the meaning of these flags; depending on the flags, an existing database is opened, or a new one created.

msilib.CreateRecord(count)
: Return a new record object by calling MSICreateRecord(). count is the number of fields of the record.

msilib.init_database(name, schema, ProductName, ProductCode, ProductVersion, Manufacturer)
: Create and return a new database name, initialize it with schema, and set the properties ProductName, ProductCode, ProductVersion, and Manufacturer.

schema must be a module object containing tables and _Validation_records attributes; typically, msilib.schema should be used.

The database will contain just the schema and the validation records when this function returns.

msilib.add_data(database, table, records)
Add all records to the table named table in database.

The table argument must be one of the predefined tables in the MSI schema, e.g. 'Feature', 'File', 'Component', 'Dialog', 'Control', etc.

records should be a list of tuples, each one containing all fields of a record according to the schema of the table. For optional fields, None can be passed.

Field values can be int or long numbers, strings, or instances of the Binary class.

class msilib.Binary(filename)
Represents entries in the Binary table; inserting such an object using add_data() reads the file named filename into the table.

msilib.add_tables(database, module)
Add all table content from module to database. module must contain an attribute tables listing all tables for which content should be added, and one attribute per table that has the actual content.

This is typically used to install the sequence tables.

msilib.add_stream(database, name, path)
Add the file path into the _Stream table of database, with the stream name name.

msilib.gen_uuid()
Return a new UUID, in the format that MSI typically requires (i.e. in curly braces, and with all hexdigits in upper-case).

See also:

FCICreateFile UuidCreate UuidToString

35.1.1 Database Objects

Database.OpenView(sql)
Return a view object, by calling MSIDatabaseOpenView(). sql is the SQL statement to execute.

Database.Commit()
Commit the changes pending in the current transaction, by calling MSIDatabaseCommit().

Database.GetSummaryInformation(count)
Return a new summary information object, by calling MsiGetSummaryInformation(). count is the maximum number of updated values.

See also:

MSIDatabaseOpenView MSIDatabaseCommit MSIGetSummaryInformation

35.1.2 View Objects

View.Execute(params)
Execute the SQL query of the view, through MSIViewExecute(). If params is not None, it is a record describing actual values of the parameter tokens in the query.

View.GetColumnInfo(kind)
: Return a record describing the columns of the view, through calling MsiViewGetColumnInfo(). kind can be either MSICOLINFO_NAMES or MSICOLINFO_TYPES.

View.Fetch()
: Return a result record of the query, through calling MsiViewFetch().

View.Modify(kind, data)
: Modify the view, by calling MsiViewModify(). kind can be one of MSIMODIFY_SEEK, MSIMODIFY_REFRESH, MSIMODIFY_INSERT, MSIMODIFY_UPDATE, MSIMODIFY_ASSIGN, MSIMODIFY_REPLACE, MSIMODIFY_MERGE, MSIMODIFY_DELETE, MSIMODIFY_INSERT_TEMPORARY, MSIMODIFY_VALIDATE, MSIMODIFY_VALIDATE_NEW, MSIMODIFY_VALIDATE_FIELD, or MSIMODIFY_VALIDATE_DELETE.

  data must be a record describing the new data.

View.Close()
: Close the view, through MsiViewClose().

See also:

MsiViewExecute MSIViewGetColumnInfo MsiViewFetch MsiViewModify MsiViewClose

35.1.3 Summary Information Objects

SummaryInformation.GetProperty(field)
: Return a property of the summary, through MsiSummaryInfoGetProperty(). field is the name of the property, and can be one of the constants PID_CODEPAGE, PID_TITLE, PID_SUBJECT, PID_AUTHOR, PID_KEYWORDS, PID_COMMENTS, PID_TEMPLATE, PID_LASTAUTHOR, PID_REVNUMBER, PID_LASTPRINTED, PID_CREATE_DTM, PID_LASTSAVE_DTM, PID_PAGECOUNT, PID_WORDCOUNT, PID_CHARCOUNT, PID_APPNAME, or PID_SECURITY.

SummaryInformation.GetPropertyCount()
: Return the number of summary properties, through MsiSummaryInfoGetPropertyCount().

SummaryInformation.SetProperty(field, value)
: Set a property through MsiSummaryInfoSetProperty(). field can have the same values as in GetProperty(), value is the new value of the property. Possible value types are integer and string.

SummaryInformation.Persist()
: Write the modified properties to the summary information stream, using MsiSummaryInfoPersist().

See also:

MsiSummaryInfoGetProperty MsiSummaryInfoGetPropertyCount MsiSummaryInfoSetProperty MsiSummaryInfoPersist

35.1.4 Record Objects

Record.GetFieldCount()
: Return the number of fields of the record, through MsiRecordGetFieldCount().

Record.GetInteger(field)
: Return the value of field as an integer where possible. field must be an integer.

Record.GetString(field)
: Return the value of field as a string where possible. field must be an integer.

Record.SetString(field, value)
Set field to value through MsiRecordSetString(). field must be an integer; value a string.

Record.SetStream(field, value)
Set field to the contents of the file named value, through MsiRecordSetStream(). field must be an integer; value a string.

Record.SetInteger(field, value)
Set field to value through MsiRecordSetInteger(). Both field and value must be an integer.

Record.ClearData()
Set all fields of the record to 0, through MsiRecordClearData().

See also:

MsiRecordGetFieldCount MsiRecordSetString MsiRecordSetStream MsiRecordSetInteger MsiRecordClear

35.1.5 Errors

All wrappers around MSI functions raise MSIError; the string inside the exception will contain more detail.

35.1.6 CAB Objects

class msilib.CAB(name)
The class CAB represents a CAB file. During MSI construction, files will be added simultaneously to the Files table, and to a CAB file. Then, when all files have been added, the CAB file can be written, then added to the MSI file.

name is the name of the CAB file in the MSI file.

append(full, file, logical)
Add the file with the pathname full to the CAB file, under the name logical. If there is already a file named logical, a new file name is created.

Return the index of the file in the CAB file, and the new name of the file inside the CAB file.

commit(database)
Generate a CAB file, add it as a stream to the MSI file, put it into the Media table, and remove the generated file from the disk.

35.1.7 Directory Objects

class msilib.Directory(database, cab, basedir, physical, logical, default[, componentflags])
Create a new directory in the Directory table. There is a current component at each point in time for the directory, which is either explicitly created through start_component(), or implicitly when files are added for the first time. Files are added into the current component, and into the cab file. To create a directory, a base directory object needs to be specified (can be None), the path to the physical directory, and a logical directory name. default specifies the DefaultDir slot in the directory table. componentflags specifies the default flags that new components get.

start_component([component[, feature[, flags[, keyfile[, uuid]]]]])
Add an entry to the Component table, and make this component the current component for this directory. If no component name is given, the directory name is used. If no feature is given, the current feature is used. If no flags are given, the directory's default flags are used. If no keyfile is given, the KeyPath is left null in the Component table.

add_file(file[, src[, version[, language]]])
: Add a file to the current component of the directory, starting a new one if there is no current component. By default, the file name in the source and the file table will be identical. If the src file is specified, it is interpreted relative to the current directory. Optionally, a version and a language can be specified for the entry in the File table.

glob(pattern[, exclude])
: Add a list of files to the current component as specified in the glob pattern. Individual files can be excluded in the exclude list.

remove_pyc()
: Remove .pyc/.pyo files on uninstall.

See also:

Directory Table File Table Component Table FeatureComponents Table

35.1.8 Features

class msilib.Feature(database, id, title, desc, display[, level=1[, parent[, directory[, attributes=0]]]])
: Add a new record to the Feature table, using the values id, parent.id, title, desc, display, level, directory, and attributes. The resulting feature object can be passed to the start_component() method of Directory.

set_current()
: Make this feature the current feature of msilib. New components are automatically added to the default feature, unless a feature is explicitly specified.

See also:

Feature Table

35.1.9 GUI classes

msilib provides several classes that wrap the GUI tables in an MSI database. However, no standard user interface is provided; use bdist_msi to create MSI files with a user-interface for installing Python packages.

class msilib.Control(dlg, name)
: Base class of the dialog controls. dlg is the dialog object the control belongs to, and name is the control's name.

event(event, argument[, condition=1[, ordering]])
: Make an entry into the ControlEvent table for this control.

mapping(event, attribute)
: Make an entry into the EventMapping table for this control.

condition(action, condition)
: Make an entry into the ControlCondition table for this control.

class msilib.RadioButtonGroup(dlg, name, property)
: Create a radio button control named name. property is the installer property that gets set when a radio button is selected.

add(name, x, y, width, height, text[, value])
: Add a radio button named name to the group, at the coordinates x, y, width, height, and with the label text. If value is omitted, it defaults to name.

class msilib.Dialog(db, name, x, y, w, h, attr, title, first, default, cancel)
Return a new Dialog object. An entry in the Dialog table is made, with the specified coordinates, dialog attributes, title, name of the first, default, and cancel controls.

control(name, type, x, y, width, height, attributes, property, text, control_next, help)
Return a new Control object. An entry in the Control table is made with the specified parameters.

This is a generic method; for specific types, specialized methods are provided.

text(name, x, y, width, height, attributes, text)
Add and return a Text control.

bitmap(name, x, y, width, height, text)
Add and return a Bitmap control.

line(name, x, y, width, height)
Add and return a Line control.

pushbutton(name, x, y, width, height, attributes, text, next_control)
Add and return a PushButton control.

radiogroup(name, x, y, width, height, attributes, property, text, next_control)
Add and return a RadioButtonGroup control.

checkbox(name, x, y, width, height, attributes, property, text, next_control)
Add and return a CheckBox control.

See also:

Dialog Table Control Table Control Types ControlCondition Table ControlEvent Table EventMapping Table RadioButton Table

35.1.10 Precomputed tables

msilib provides a few subpackages that contain only schema and table definitions. Currently, these definitions are based on MSI version 2.0.

msilib.schema
This is the standard MSI schema for MSI 2.0, with the tables variable providing a list of table definitions, and _Validation_records providing the data for MSI validation.

msilib.sequence
This module contains table contents for the standard sequence tables: AdminExecuteSequence, AdminUISequence, AdvtExecuteSequence, InstallExecuteSequence, and InstallUISequence.

msilib.text
This module contains definitions for the UIText and ActionText tables, for the standard installer actions.

35.2 msvcrt — Useful routines from the MS VC++ runtime

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the getpass module uses this in the implementation of the getpass() function.

Further documentation on these functions can be found in the Platform API documentation.

The module implements both the normal and wide char variants of the console I/O api. The normal API deals only with ASCII characters and is of limited use for internationalized applications. The wide char API should be used where ever possible.

35.2.1 File Operations

msvcrt.locking(fd, mode, nbytes)
: Lock part of a file based on file descriptor fd from the C runtime. Raises IOError on failure. The locked region of the file extends from the current file position for nbytes bytes, and may continue beyond the end of the file. mode must be one of the LK_* constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged; they must be unlocked individually.

msvcrt.LK_LOCK
msvcrt.LK_RLCK
: Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, IOError is raised.

msvcrt.LK_NBLCK
msvcrt.LK_NBRLCK
: Locks the specified bytes. If the bytes cannot be locked, IOError is raised.

msvcrt.LK_UNLCK
: Unlocks the specified bytes, which must have been previously locked.

msvcrt.setmode(fd, flags)
: Set the line-end translation mode for the file descriptor fd. To set it to text mode, flags should be os.O_TEXT; for binary, it should be os.O_BINARY.

msvcrt.open_osfhandle(handle, flags)
: Create a C runtime file descriptor from the file handle handle. The flags parameter should be a bitwise OR of os.O_APPEND, os.O_RDONLY, and os.O_TEXT. The returned file descriptor may be used as a parameter to os.fdopen() to create a file object.

msvcrt.get_osfhandle(fd)
: Return the file handle for the file descriptor fd. Raises IOError if fd is not recognized.

35.2.2 Console I/O

msvcrt.kbhit()
: Return true if a keypress is waiting to be read.

msvcrt.getch()
: Read a keypress and return the resulting character. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for Enter to be pressed. If the pressed key was a special function key, this will return `'\000'` or `'\xe0'`; the next call will return the keycode. The Control-C keypress cannot be read with this function.

msvcrt.getwch()
: Wide char variant of getch(), returning a Unicode value.

  New in version 2.6.

msvcrt.getche()
: Similar to getch(), but the keypress will be echoed if it represents a printable character.

msvcrt.getwche()
Wide char variant of getche(), returning a Unicode value.

New in version 2.6.

msvcrt.putch(char)
Print the character char to the console without buffering.

msvcrt.putwch(unicode_char)
Wide char variant of putch(), accepting a Unicode value.

New in version 2.6.

msvcrt.ungetch(char)
Cause the character char to be "pushed back" into the console buffer; it will be the next character read by getch() or getche().

msvcrt.ungetwch(unicode_char)
Wide char variant of ungetch(), accepting a Unicode value.

New in version 2.6.

35.2.3 Other Functions

msvcrt.heapmin()
Force the malloc() heap to clean itself up and return unused blocks to the operating system. On failure, this raises IOError.

35.3 _winreg — Windows registry access

Note: The _winreg module has been renamed to winreg in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python 3.

New in version 2.0.

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a handle object is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

This module offers the following functions:

_winreg.CloseKey(hkey)
Closes a previously opened registry key. The hkey argument specifies a previously opened key.

Note: If hkey is not closed using this method (or via hkey.Close()), it is closed when the hkey object is destroyed by Python.

_winreg.ConnectRegistry(computer_name, key)
Establishes a connection to a predefined registry handle on another computer, and returns a handle object.

computer_name is the name of the remote computer, of the form r"\\computername". If None, the local computer is used.

key is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, a WindowsError exception is raised.

_winreg.CreateKey(key, sub_key)
Creates or opens the specified key, returning a handle object.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that names the key this method opens or creates.

If key is one of the predefined keys, sub_key may be None. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, a WindowsError exception is raised.

_winreg.CreateKeyEx(key, sub_key[, res[, sam]])
Creates or opens the specified key, returning a handle object.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that names the key this method opens or creates.

res is a reserved integer, and must be zero. The default is zero.

sam is an integer that specifies an access mask that describes the desired security access for the key. Default is KEY_ALL_ACCESS. See Access Rights for other allowed values.

If key is one of the predefined keys, sub_key may be None. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, a WindowsError exception is raised.

New in version 2.7.

_winreg.DeleteKey(key, sub_key)
Deletes the specified key.

key is an already open key, or any one of the predefined HKEY_* constants.

sub_key is a string that must be a subkey of the key identified by the key parameter. This value must not be None, and the key may not have subkeys.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, a WindowsError exception is raised.

_winreg.DeleteKeyEx(key, sub_key[, sam[, res]])
Deletes the specified key.

Note: The DeleteKeyEx() function is implemented with the RegDeleteKeyEx Windows API function, which is specific to 64-bit versions of Windows. See the RegDeleteKeyEx documentation.

---

key is an already open key, or any one of the predefined HKEY_* constants.

sub_key is a string that must be a subkey of the key identified by the key parameter. This value must not be None, and the key may not have subkeys.

res is a reserved integer, and must be zero. The default is zero.

sam is an integer that specifies an access mask that describes the desired security access for the key. Default is KEY_WOW64_64KEY. See Access Rights for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, a WindowsError exception is raised.

On unsupported Windows versions, NotImplementedError is raised.

New in version 2.7.

_winreg.DeleteValue(key, value)
Removes a named value from a registry key.

key is an already open key, or one of the predefined HKEY_* constants.

value is a string that identifies the value to remove.

_winreg.EnumKey(key, index)
Enumerates subkeys of an open registry key, returning a string.

key is an already open key, or any one of the predefined HKEY_* constants.

index is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until a WindowsError exception is raised, indicating, no more values are available.

_winreg.EnumValue(key, index)
Enumerates values of an open registry key, returning a tuple.

key is an already open key, or any one of the predefined HKEY_* constants.

index is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until a WindowsError exception is raised, indicating no more values.

The result is a tuple of 3 items:

Index	Meaning
0	A string that identifies the value name
1	An object that holds the value data, and whose type depends on the underlying registry type
2	An integer that identifies the type of the value data (see table in docs for SetValueEx())

_winreg.ExpandEnvironmentStrings(unicode)
Expands environment variable placeholders %NAME% in unicode strings like REG_EXPAND_SZ:

```
>>> ExpandEnvironmentStrings(u"%windir%")
u"C:\\Windows"
```

New in version 2.6.

_winreg.FlushKey(key)
Writes all the attributes of a key to the registry.

key is an already open key, or one of the predefined HKEY_* constants.

It is not necessary to call FlushKey() to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike CloseKey(), the FlushKey() method returns only when all the data has been written to the registry. An application should only call FlushKey() if it requires absolute certainty that registry changes are on disk.

Note: If you don't know whether a FlushKey() call is required, it probably isn't.

_winreg.LoadKey(key, sub_key, file_name)
Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

key is a handle returned by ConnectRegistry() or one of the constants HKEY_USERS or HKEY_LOCAL_MACHINE.

sub_key is a string that identifies the subkey to load.

file_name is the name of the file to load registry data from. This file must have been created with the SaveKey() function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to LoadKey() fails if the calling process does not have the SE_RESTORE_PRIVILEGE privilege. Note that privileges are different from permissions – see the RegLoadKey documentation for more details.

If key is a handle returned by ConnectRegistry(), then the path specified in file_name is relative to the remote computer.

_winreg.OpenKey(key, sub_key[, res[, sam]])
Opens the specified key, returning a handle object.

key is an already open key, or any one of the predefined HKEY_* constants.

sub_key is a string that identifies the sub_key to open.

res is a reserved integer, and must be zero. The default is zero.

sam is an integer that specifies an access mask that describes the desired security access for the key. Default is KEY_READ. See Access Rights for other allowed values.

The result is a new handle to the specified key.

If the function fails, WindowsError is raised.

_winreg.OpenKeyEx()
The functionality of OpenKeyEx() is provided via OpenKey(), by the use of default arguments.

_winreg.QueryInfoKey(key)
Returns information about a key, as a tuple.

key is an already open key, or one of the predefined HKEY_* constants.

The result is a tuple of 3 items:

Index	Meaning
0	An integer giving the number of sub keys this key has.
1	An integer giving the number of values this key has.
2	A long integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1601.

_winreg.QueryValue(key, sub_key)
Retrieves the unnamed value for a key, as a string.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that holds the name of the subkey with which the value is associated. If this parameter is None or empty, the function retrieves the value set by the SetValue() method for the key identified by key.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a NULL name. But the underlying API call doesn't return the type, so always use QueryValueEx() if possible.

_winreg.QueryValueEx(key, value_name)
Retrieves the type and data for a specified value name associated with an open registry key.

key is an already open key, or one of the predefined HKEY_* constants.

value_name is a string indicating the value to query.

The result is a tuple of 2 items:

Index	Meaning
0	The value of the registry item.
1	An integer giving the registry type for this value (see table in docs for SetValueEx())

_winreg.SaveKey(key, file_name)
Saves the specified key, and all its subkeys to the specified file.

key is an already open key, or one of the predefined HKEY_* constants.

file_name is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the LoadKey() method.

If key represents a key on a remote computer, the path described by file_name is relative to the remote computer. The caller of this method must possess the SeBackupPrivilege security privilege. Note that privileges are different than permissions – see the Conflicts Between User Rights and Permissions documentation for more details.

This function passes NULL for security_attributes to the API.

_winreg.SetValue(key, sub_key, type, value)
Associates a value with a specified key.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. Currently this must be REG_SZ, meaning only strings are supported. Use the SetValueEx() function for support for other data types.

value is a string that specifies the new value.

If the key specified by the sub_key parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the key parameter must have been opened with KEY_SET_VALUE access.

_winreg.SetValueEx(key, value_name, reserved, type, value)
Stores data in the value field of an open registry key.

key is an already open key, or one of the predefined HKEY_* constants.

value_name is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. See Value Types for the available types.

reserved can be anything – zero is always passed to the API.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with KEY_SET_VALUE access.

To open the key, use the CreateKey() or OpenKey() methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

_winreg.DisableReflectionKey(key)
Disables registry reflection for 32-bit processes running on a 64-bit operating system.

key is an already open key, or one of the predefined HKEY_* constants.

Will generally raise NotImplemented if executed on a 32-bit operating system.

If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

_winreg.EnableReflectionKey(key)
Restores registry reflection for the specified disabled key.

key is an already open key, or one of the predefined HKEY_* constants.

Will generally raise NotImplemented if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

_winreg.QueryReflectionKey(key)
Determines the reflection state for the specified key.

key is an already open key, or one of the predefined HKEY_* constants.

Returns True if reflection is disabled.

Will generally raise NotImplemented if executed on a 32-bit operating system.

35.3.1 Constants

The following constants are defined for use in many _winreg functions.

HKEY_* Constants

_winreg.HKEY_CLASSES_ROOT
Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

_winreg.HKEY_CURRENT_USER
Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

_winreg.HKEY_LOCAL_MACHINE
Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

_winreg.HKEY_USERS
Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

_winreg.HKEY_PERFORMANCE_DATA
Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.

_winreg.HKEY_CURRENT_CONFIG
Contains information about the current hardware profile of the local computer system.

_winreg.HKEY_DYN_DATA
This key is not used in versions of Windows after 98.

Access Rights

For more information, see Registry Key Security and Access.

_winreg.KEY_ALL_ACCESS
Combines the STANDARD_RIGHTS_REQUIRED, KEY_QUERY_VALUE, KEY_SET_VALUE, KEY_CREATE_SUB_KEY, KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, and KEY_CREATE_LINK access rights.

_winreg.KEY_WRITE
Combines the STANDARD_RIGHTS_WRITE, KEY_SET_VALUE, and KEY_CREATE_SUB_KEY access rights.

_winreg.KEY_READ
Combines the STANDARD_RIGHTS_READ, KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY values.

_winreg.KEY_EXECUTE
Equivalent to KEY_READ.

_winreg.KEY_QUERY_VALUE
Required to query the values of a registry key.

_winreg.KEY_SET_VALUE
Required to create, delete, or set a registry value.

_winreg.KEY_CREATE_SUB_KEY
Required to create a subkey of a registry key.

_winreg.KEY_ENUMERATE_SUB_KEYS
Required to enumerate the subkeys of a registry key.

_winreg.KEY_NOTIFY
Required to request change notifications for a registry key or for subkeys of a registry key.

_winreg.KEY_CREATE_LINK
Reserved for system use.

64-bit Specific

For more information, see Accessing an Alternate Registry View.

_winreg.KEY_WOW64_64KEY
Indicates that an application on 64-bit Windows should operate on the 64-bit registry view.

_winreg.KEY_WOW64_32KEY
Indicates that an application on 64-bit Windows should operate on the 32-bit registry view.

Value Types

For more information, see Registry Value Types.

_winreg.REG_BINARY
Binary data in any form.

_winreg.REG_DWORD
32-bit number.

_winreg.REG_DWORD_LITTLE_ENDIAN
A 32-bit number in little-endian format.

_winreg.REG_DWORD_BIG_ENDIAN
A 32-bit number in big-endian format.

_winreg.REG_EXPAND_SZ
Null-terminated string containing references to environment variables (%PATH%).

_winreg.REG_LINK
A Unicode symbolic link.

_winreg.REG_MULTI_SZ
A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

_winreg.REG_NONE
No defined value type.

_winreg.REG_RESOURCE_LIST
A device-driver resource list.

_winreg.REG_FULL_RESOURCE_DESCRIPTOR
A hardware setting.

_winreg.REG_RESOURCE_REQUIREMENTS_LIST
A hardware resource list.

_winreg.REG_SZ
A null-terminated string.

35.3.2 Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the Close() method on the object, or the CloseKey() function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for __nonzero__() – thus:

```
if handle:
    print "Yes"
```

will print Yes if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in int() function), in which case the underlying Windows handle value is returned. You can also use the Detach() method to return the integer handle, and also disconnect the Windows handle from the handle object.

PyHKEY.Close()
: Closes the underlying Windows handle.

  If the handle is already closed, no error is raised.

PyHKEY.Detach()
: Detaches the Windows handle from the handle object.

  The result is an integer (or long on 64 bit Windows) that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

  After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

PyHKEY.__enter__()
PyHKEY.__exit__(*exc_info)
: The HKEY object implements __enter__() and __exit__() and thus supports the context protocol for the with statement:

  ```
  with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
      ... # work with key
  ```

  will automatically close key when control leaves the with block.

  New in version 2.6.

35.4 winsound — Sound-playing interface for Windows

New in version 1.5.2.

The winsound module provides access to the basic sound-playing machinery provided by Windows platforms. It includes functions and several constants.

winsound.Beep(frequency, duration)
: Beep the PC's speaker. The frequency parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The duration parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, RuntimeError is raised.

  New in version 1.6.

winsound.PlaySound(sound, flags)
: Call the underlying PlaySound() function from the Platform API. The sound parameter may be a filename, audio data as a string, or None. Its interpretation depends on the value of flags, which can be a bitwise ORed combination of the constants described below. If the sound parameter is None, any currently playing waveform sound is stopped. If the system indicates an error, RuntimeError is raised.

winsound.MessageBeep([type=MB_OK])
Call the underlying MessageBeep() function from the Platform API. This plays a sound as specified in the registry. The type argument specifies which sound to play; possible values are -1, MB_ICONASTERISK, MB_ICONEXCLAMATION, MB_ICONHAND, MB_ICONQUESTION, and MB_OK, all described below. The value -1 produces a "simple beep"; this is the final fallback if a sound cannot be played otherwise.

New in version 2.3.

winsound.SND_FILENAME
The sound parameter is the name of a WAV file. Do not use with SND_ALIAS.

winsound.SND_ALIAS
The sound parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless SND_NODEFAULT is also specified. If no default sound is registered, raise RuntimeError. Do not use with SND_FILENAME.

All Win32 systems support at least the following; most systems support many more:

PlaySound() name	Corresponding Control Panel Sound name
'SystemAsterisk'	Asterisk
'SystemExclamation'	Exclamation
'SystemExit'	Exit Windows
'SystemHand'	Critical Stop
'SystemQuestion'	Question

For example:

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.SND_LOOP
Play the sound repeatedly. The SND_ASYNC flag must also be used to avoid blocking. Cannot be used with SND_MEMORY.

winsound.SND_MEMORY
The sound parameter to PlaySound() is a memory image of a WAV file, as a string.

Note: This module does not support playing from a memory image asynchronously, so a combination of this flag and SND_ASYNC will raise RuntimeError.

---

winsound.SND_PURGE
Stop playing all instances of the specified sound.

Note: This flag is not supported on modern Windows platforms.

---

winsound.SND_ASYNC
Return immediately, allowing sounds to play asynchronously.

winsound.SND_NODEFAULT
: If the specified sound cannot be found, do not play the system default sound.

winsound.SND_NOSTOP
: Do not interrupt sounds currently playing.

winsound.SND_NOWAIT
: Return immediately if the sound driver is busy.

Note: This flag is not supported on modern Windows platforms.

---

winsound.MB_ICONASTERISK
: Play the SystemDefault sound.

winsound.MB_ICONEXCLAMATION
: Play the SystemExclamation sound.

winsound.MB_ICONHAND
: Play the SystemHand sound.

winsound.MB_ICONQUESTION
: Play the SystemQuestion sound.

winsound.MB_OK
: Play the SystemDefault sound.

CHAPTER

THIRTYSIX

UNIX SPECIFIC SERVICES

The modules described in this chapter provide interfaces to features that are unique to the Unix operating system, or in some cases to some or many variants of it. Here's an overview:

36.1 posix — The most common POSIX system calls

This module provides access to operating system functionality that is standardized by the C Standard and the POSIX standard (a thinly disguised Unix interface).

Do not import this module directly. Instead, import the module os, which provides a portable version of this interface. On Unix, the os module provides a superset of the posix interface. On non-Unix operating systems the posix module is not available, but a subset is always available through the os interface. Once os is imported, there is no performance penalty in using it instead of posix. In addition, os provides some additional functionality, such as automatically calling putenv() when an entry in os.environ is changed.

Errors are reported as exceptions; the usual exceptions are given for type errors, while errors reported by the system calls raise OSError.

36.1.1 Large File Support

Several operating systems (including AIX, HP-UX, Irix and Solaris) provide support for files that are larger than 2 GB from a C programming model where int and long are 32-bit values. This is typically accomplished by defining the relevant size and offset types as 64-bit values. Such files are sometimes referred to as large files.

Large file support is enabled in Python when the size of an off_t is larger than a long and the long long type is available and is at least as large as an off_t. Python longs are then used to represent file sizes, offsets and other values that can exceed the range of a Python int. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, it is enabled by default with recent versions of Irix, but with Solaris 2.6 and 2.7 you need to do something like:

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
        ./configure
```

On large-file-capable Linux systems, this might work:

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
        ./configure
```

36.1.2 Notable Module Contents

In addition to many functions described in the os module documentation, posix defines the following data item:

posix.environ
A dictionary representing the string environment at the time the interpreter was started. For example, environ['HOME'] is the pathname of your home directory, equivalent to getenv("HOME") in C.

Modifying this dictionary does not affect the string environment passed on by execv(), popen() or system(); if you need to change the environment, pass environ to execve() or add variable assignments and export statements to the command string for system() or popen().

Note: The os module provides an alternate implementation of environ which updates the environment on modification. Note also that updating os.environ will render this dictionary obsolete. Use of the os module version of this is recommended over direct access to the posix module.

36.2 pwd — The password database

This module provides access to the Unix user account and password database. It is available on all Unix versions.

Password database entries are reported as a tuple-like object, whose attributes correspond to the members of the passwd structure (Attribute field below, see <pwd.h>):

Index	Attribute	Meaning
0	pw_name	Login name
1	pw_passwd	Optional encrypted password
2	pw_uid	Numerical user ID
3	pw_gid	Numerical group ID
4	pw_gecos	User name or comment field
5	pw_dir	User home directory
6	pw_shell	User command interpreter

The uid and gid items are integers, all others are strings. KeyError is raised if the entry asked for cannot be found.

Note: In traditional Unix the field pw_passwd usually contains a password encrypted with a DES derived algorithm (see module crypt). However most modern unices use a so-called shadow password system. On those unices the pw_passwd field only contains an asterisk ('*') or the letter 'x' where the encrypted password is stored in a file /etc/shadow which is not world readable. Whether the pw_passwd field contains anything useful is system-dependent. If available, the spwd module should be used where access to the encrypted password is required.

It defines the following items:

pwd.getpwuid(uid)
Return the password database entry for the given numeric user ID.

pwd.getpwnam(name)
Return the password database entry for the given user name.

pwd.getpwall()
Return a list of all available password database entries, in arbitrary order.

See also:

Module grp An interface to the group database, similar to this.

Module spwd An interface to the shadow password database, similar to this.

36.3 spwd — The shadow password database

New in version 2.5.

This module provides access to the Unix shadow password database. It is available on various Unix versions.

You must have enough privileges to access the shadow password database (this usually means you have to be root).

Shadow password database entries are reported as a tuple-like object, whose attributes correspond to the members of the spwd structure (Attribute field below, see <shadow.h>):

Index	Attribute	Meaning
0	sp_nam	Login name
1	sp_pwd	Encrypted password
2	sp_lstchg	Date of last change
3	sp_min	Minimal number of days between changes
4	sp_max	Maximum number of days between changes
5	sp_warn	Number of days before password expires to warn user about it
6	sp_inact	Number of days after password expires until account is blocked
7	sp_expire	Number of days since 1970-01-01 until account is disabled
8	sp_flag	Reserved

The sp_nam and sp_pwd items are strings, all others are integers. KeyError is raised if the entry asked for cannot be found.

It defines the following items:

spwd.getspnam(name)
Return the shadow password database entry for the given user name.

spwd.getspall()
Return a list of all available shadow password database entries, in arbitrary order.

See also:

Module grp An interface to the group database, similar to this.

Module pwd An interface to the normal password database, similar to this.

36.4 grp — The group database

This module provides access to the Unix group database. It is available on all Unix versions.

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the group structure (Attribute field below, see <pwd.h>):

Index	Attribute	Meaning
0	gr_name	the name of the group
1	gr_passwd	the (encrypted) group password; often empty
2	gr_gid	the numerical group ID
3	gr_mem	all the group member's user names

The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database. Check both databases to get complete membership information. Also note that a gr_name that starts with a + or - is likely to be a YP/NIS reference and may not be accessible via getgrnam() or getgrgid().)

It defines the following items:

grp.getgrgid(gid)
: Return the group database entry for the given numeric group ID. KeyError is raised if the entry asked for cannot be found.

grp.getgrnam(name)
: Return the group database entry for the given group name. KeyError is raised if the entry asked for cannot be found.

grp.getgrall()
: Return a list of all available group entries, in arbitrary order.

See also:

Module pwd An interface to the user database, similar to this.

Module spwd An interface to the shadow password database, similar to this.

36.5 crypt — Function to check Unix passwords

This module implements an interface to the crypt(3) routine, which is a one-way hash function based upon a modified DES algorithm; see the Unix man page for further details. Possible uses include allowing Python scripts to accept typed passwords from the user, or attempting to crack Unix passwords with a dictionary.

Notice that the behavior of this module depends on the actual implementation of the crypt(3) routine in the running system. Therefore, any extensions available on the current implementation will also be available on this module.

crypt.crypt(word, salt)
: word will usually be a user's password as typed at a prompt or in a graphical interface. salt is usually a random two-character string which will be used to perturb the DES algorithm in one of 4096 ways. The characters in salt must be in the set [./a-zA-Z0-9]. Returns the hashed password as a string, which will be composed of characters from the same alphabet as the salt (the first two characters represent the salt itself).

  Since a few crypt(3) extensions allow different values, with different sizes in the salt, it is recommended to use the full crypted password as salt when checking for a password.

A simple example illustrating typical use:

```
import crypt, getpass, pwd

def login():
    username = raw_input('Python login:')
```

(continues on next page)

(continued from previous page)

```
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise NotImplementedError(
                "Sorry, currently no support for shadow passwords")
        cleartext = getpass.getpass()
        return crypt.crypt(cleartext, cryptedpasswd) == cryptedpasswd
    else:
        return 1
```

36.6 dl — Call C functions in shared objects

Deprecated since version 2.6: The dl module has been removed in Python 3. Use the ctypes module instead.

The dl module defines an interface to the dlopen() function, which is the most common interface on Unix platforms for handling dynamically linked libraries. It allows the program to call arbitrary functions in such a library.

> Warning: The dl module bypasses the Python type system and error handling. If used incorrectly it may cause segmentation faults, crashes or other incorrect behaviour.

Note: This module will not work unless sizeof(int) == sizeof(long) == sizeof(char *) If this is not the case, SystemError will be raised on import.

The dl module defines the following function:

dl.open(name[, mode=RTLD_LAZY])
: Open a shared object file, and return a handle. Mode signifies late binding (RTLD_LAZY) or immediate binding (RTLD_NOW). Default is RTLD_LAZY. Note that some systems do not support RTLD_NOW.

  Return value is a dlobject.

The dl module defines the following constants:

dl.RTLD_LAZY
: Useful as an argument to open().

dl.RTLD_NOW
: Useful as an argument to open(). Note that on systems which do not support immediate binding, this constant will not appear in the module. For maximum portability, use hasattr() to determine if the system supports immediate binding.

The dl module defines the following exception:

exception dl.error
: Exception raised when an error has occurred inside the dynamic loading and linking routines.

Example:

```
>>> import dl, time
>>> a=dl.open('/lib/libc.so.6')
```

(continues on next page)

(continued from previous page)

```
>>> a.call('time'), time.time()
(929723914, 929723914.498)
```

This example was tried on a Debian GNU/Linux system, and is a good example of the fact that using this module is usually a bad alternative.

36.6.1 Dl Objects

Dl objects, as returned by open() above, have the following methods:

dl.close()
: Free all resources, except the memory.

dl.sym(name)
: Return the pointer for the function named name, as a number, if it exists in the referenced shared object, otherwise None. This is useful in code like:

```
>>> if a.sym('time'):
...     a.call('time')
... else:
...     time.time()
```

(Note that this function will return a non-zero number, as zero is the NULL pointer)

dl.call(name[, arg1[, arg2...]])
: Call the function named name in the referenced shared object. The arguments must be either Python integers, which will be passed as is, Python strings, to which a pointer will be passed, or None, which will be passed as NULL. Note that strings should only be passed to functions as const char*, as Python will not like its string mutated.

There must be at most 10 arguments, and arguments not given will be treated as None. The function's return value must be a C long, which is a Python integer.

36.7 termios — POSIX style tty control

This module provides an interface to the POSIX calls for tty I/O control. For a complete description of these calls, see termios(2) Unix manual page. It is only available for those Unix versions that support POSIX termios style tty I/O control configured during installation.

All functions in this module take a file descriptor fd as their first argument. This can be an integer file descriptor, such as returned by sys.stdin.fileno(), or a file object, such as sys.stdin itself.

This module also defines all the constants needed to work with the functions provided here; these have the same name as their counterparts in C. Please refer to your system documentation for more information on using these terminal control interfaces.

The module defines the following functions:

termios.tcgetattr(fd)
: Return a list containing the tty attributes for file descriptor fd, as follows: [iflag, oflag, cflag, lflag, ispeed, ospeed, cc] where cc is a list of the tty special characters (each a string of length 1, except the items with indices VMIN and VTIME, which are integers when these fields are defined). The interpretation of the flags and the speeds as well as the indexing in the cc array must be done using the symbolic constants defined in the termios module.

termios.tcsetattr(fd, when, attributes)
: Set the tty attributes for file descriptor fd from the attributes, which is a list like the one returned by tcgetattr(). The when argument determines when the attributes are changed: TCSANOW to change immediately, TCSADRAIN to change after transmitting all queued output, or TCSAFLUSH to change after transmitting all queued output and discarding all queued input.

termios.tcsendbreak(fd, duration)
: Send a break on file descriptor fd. A zero duration sends a break for 0.25 –0.5 seconds; a nonzero duration has a system dependent meaning.

termios.tcdrain(fd)
: Wait until all output written to file descriptor fd has been transmitted.

termios.tcflush(fd, queue)
: Discard queued data on file descriptor fd. The queue selector specifies which queue: TCIFLUSH for the input queue, TCOFLUSH for the output queue, or TCIOFLUSH for both queues.

termios.tcflow(fd, action)
: Suspend or resume input or output on file descriptor fd. The action argument can be TCOOFF to suspend output, TCOON to restart output, TCIOFF to suspend input, or TCION to restart input.

See also:

Module tty Convenience functions for common terminal control operations.

36.7.1 Example

Here's a function that prompts for a password with echoing turned off. Note the technique using a separate tcgetattr() call and a try ... finally statement to ensure that the old tty attributes are restored exactly no matter what happens:

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = raw_input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

36.8 tty — Terminal control functions

The tty module defines functions for putting the tty into cbreak and raw modes.

Because it requires the termios module, it will work only on Unix.

The tty module defines the following functions:

tty.setraw(fd[, when])
: Change the mode of the file descriptor fd to raw. If when is omitted, it defaults to termios.TCSAFLUSH, and is passed to termios.tcsetattr().

tty.setcbreak(fd[, when])
: Change the mode of file descriptor fd to cbreak. If when is omitted, it defaults to termios.TCSAFLUSH, and is passed to termios.tcsetattr().

See also:

Module termios Low-level terminal control interface.

36.9 pty — Pseudo-terminal utilities

The pty module defines operations for handling the pseudo-terminal concept: starting another process and being able to write to and read from its controlling terminal programmatically.

Because pseudo-terminal handling is highly platform dependent, there is code to do it only for Linux. (The Linux code is supposed to work on other platforms, but hasn't been tested yet.)

The pty module defines the following functions:

pty.fork()
: Fork. Connect the child's controlling terminal to a pseudo-terminal. Return value is (pid, fd). Note that the child gets pid 0, and the fd is invalid. The parent's return value is the pid of the child, and fd is a file descriptor connected to the child's controlling terminal (and also to the child's standard input and output).

pty.openpty()
: Open a new pseudo-terminal pair, using os.openpty() if possible, or emulation code for generic Unix systems. Return a pair of file descriptors (master, slave), for the master and the slave end, respectively.

pty.spawn(argv[, master_read[, stdin_read]])
: Spawn a process, and connect its controlling terminal with the current process's standard io. This is often used to baffle programs which insist on reading from the controlling terminal.

  The functions master_read and stdin_read should be functions which read from a file descriptor. The defaults try to read 1024 bytes each time they are called.

36.10 fcntl — The fcntl and ioctl system calls

This module performs file control and I/O control on file descriptors. It is an interface to the fcntl() and ioctl() Unix routines. For a complete description of these calls, see fcntl(2) and ioctl(2) Unix manual pages.

All functions in this module take a file descriptor fd as their first argument. This can be an integer file descriptor, such as returned by sys.stdin.fileno(), or a file object, such as sys.stdin itself, which provides a fileno() which returns a genuine file descriptor.

The module defines the following functions:

fcntl.fcntl(fd, op[, arg])
: Perform the operation op on file descriptor fd (file objects providing a fileno() method are accepted as well). The values used for for op are operating system dependent, and are available as constants in the fcntl module, using the same names as used in the relevant C header files. The argument arg is optional, and defaults to the integer value 0. When present, it can either be an integer value, or a string. With the argument missing or an integer value, the return value of this function is the integer return value of the C fcntl() call. When the argument is a string it represents a binary structure, e.g. created by struct.pack(). The binary data is copied to a buffer whose address is passed to the C fcntl() call. The return value after a successful call is the contents of the buffer, converted to a string object. The length of the returned string will be the same as the length of the arg argument. This is limited

to 1024 bytes. If the information returned in the buffer by the operating system is larger than 1024 bytes, this is most likely to result in a segmentation violation or a more subtle data corruption.

If the fcntl() fails, an IOError is raised.

fcntl.ioctl(fd, op[, arg[, mutate_flag]])
This function is identical to the fcntl() function, except that the operations are typically defined in the library module termios and the argument handling is even more complicated.

The op parameter is limited to values that can fit in 32-bits. Additional constants of interest for use as the op argument can be found in the termios module, under the same names as used in the relevant C header files.

The parameter arg can be one of an integer, absent (treated identically to the integer 0), an object supporting the read-only buffer interface (most likely a plain Python string) or an object supporting the read-write buffer interface.

In all but the last case, behaviour is as for the fcntl() function.

If a mutable buffer is passed, then the behaviour is determined by the value of the mutate_flag parameter.

If it is false, the buffer's mutability is ignored and behaviour is as for a read-only buffer, except that the 1024 byte limit mentioned above is avoided – so long as the buffer you pass is as least as long as what the operating system wants to put there, things should work.

If mutate_flag is true, then the buffer is (in effect) passed to the underlying ioctl() system call, the latter's return code is passed back to the calling Python, and the buffer's new contents reflect the action of the ioctl(). This is a slight simplification, because if the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to ioctl() and copied back into the supplied buffer.

If mutate_flag is not supplied, then from Python 2.5 it defaults to true, which is a change from versions 2.3 and 2.4. Supply the argument explicitly if version portability is a priority.

If the ioctl() fails, an IOError exception is raised.

An example:

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

fcntl.flock(fd, op)
Perform the lock operation op on file descriptor fd (file objects providing a fileno() method are accepted as well). See the Unix manual flock(2) for details. (On some systems, this function is emulated using fcntl().)

If the flock() fails, an IOError exception is raised.

fcntl.lockf(fd, operation[, length[, start[, whence]]])
This is essentially a wrapper around the fcntl() locking calls. fd is the file descriptor of the file to lock or unlock, and operation is one of the following values:

- LOCK_UN – unlock

- LOCK_SH – acquire a shared lock
- LOCK_EX – acquire an exclusive lock

When operation is LOCK_SH or LOCK_EX, it can also be bitwise ORed with LOCK_NB to avoid blocking on lock acquisition. If LOCK_NB is used and the lock cannot be acquired, an IOError will be raised and the exception will have an errno attribute set to EACCES or EAGAIN (depending on the operating system; for portability, check for both values). On at least some systems, LOCK_EX can only be used if the file descriptor refers to a file opened for writing.

length is the number of bytes to lock, start is the byte offset at which the lock starts, relative to whence, and whence is as with io.IOBase.seek(), specifically:

- 0 – relative to the start of the file (os.SEEK_SET)
- 1 – relative to the current buffer position (os.SEEK_CUR)
- 2 – relative to the end of the file (os.SEEK_END)

The default for start is 0, which means to start at the beginning of the file. The default for length is 0 which means to lock to the end of the file. The default for whence is also 0.

Examples (all on a SVR4 compliant system):

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

Note that in the first example the return value variable rv will hold an integer value; in the second example it will hold a string value. The structure lay-out for the lockdata variable is system dependent — therefore using the flock() call may be better.

See also:

Module os If the locking flags O_SHLOCK and O_EXLOCK are present in the os module (on BSD only), the os.open() function provides an alternative to the lockf() and flock() functions.

36.11 pipes — Interface to shell pipelines

Source code: Lib/pipes.py

The pipes module defines a class to abstract the concept of a pipeline — a sequence of converters from one file to another.

Because the module uses /bin/sh command lines, a POSIX or compatible shell for os.system() and os.popen() is required.

class pipes.Template
An abstraction of a pipeline.

Example:

```
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```

pipes.quote(s)
: Deprecated since version 2.7: Prior to Python 2.7, this function was not publicly documented. It is finally exposed publicly in Python 3.3 as the quote function in the shlex module.

  Return a shell-escaped version of the string s. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

  This idiom would be unsafe:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print command  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

  quote() lets you plug the security hole:

```
>>> command = 'ls -l {}'.format(quote(filename))
>>> print command
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print remote_command
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

  The quoting is compatible with UNIX shells and with shlex.split():

```
>>> remote_command = shlex.split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = shlex.split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

36.11.1 Template Objects

Template objects following methods:

Template.reset()
: Restore a pipeline template to its initial state.

Template.clone()
: Return a new, equivalent, pipeline template.

Template.debug(flag)
: If flag is true, turn debugging on. Otherwise, turn debugging off. When debugging is on, commands to be executed are printed, and the shell is given set -x command to be more verbose.

Template.append(cmd, kind)
: Append a new action at the end. The cmd variable must be a valid bourne shell command. The kind variable consists of two letters.

The first letter can be either of `'-'` (which means the command reads its standard input), `'f'` (which means the commands reads a given file on the command line) or `'.'` (which means the commands reads no input, and hence must be first.)

Similarly, the second letter can be either of `'-'` (which means the command writes to standard output), `'f'` (which means the command writes a file on the command line) or `'.'` (which means the command does not write anything, and hence must be last.)

Template.prepend(cmd, kind)
Add a new action at the beginning. See append() for explanations of the arguments.

Template.open(file, mode)
Return a file-like object, open to file, but read from or written to by the pipeline. Note that only one of `'r'`, `'w'` may be given.

Template.copy(infile, outfile)
Copy infile to outfile through the pipe.

36.12 posixfile — File-like objects with locking support

Deprecated since version 1.5: The locking operation that this module provides is done better and more portably by the fcntl.lockf() call.

This module implements some additional functionality over the built-in file objects. In particular, it implements file locking, control over the file flags, and an easy interface to duplicate the file object. The module defines a new file object, the posixfile object. It has all the standard file object methods and adds the methods described below. This module only works for certain flavors of Unix, since it uses fcntl.fcntl() for file locking.

To instantiate a posixfile object, use the posixfile.open() function. The resulting object looks and feels roughly the same as a standard file object.

The posixfile module defines the following constants:

posixfile.SEEK_SET
Offset is calculated from the start of the file.

posixfile.SEEK_CUR
Offset is calculated from the current position in the file.

posixfile.SEEK_END
Offset is calculated from the end of the file.

The posixfile module defines the following functions:

posixfile.open(filename[, mode[, bufsize]])
Create a new posixfile object with the given filename and mode. The filename, mode and bufsize arguments are interpreted the same way as by the built-in open() function.

posixfile.fileopen(fileobject)
Create a new posixfile object with the given standard file object. The resulting object has the same filename and mode as the original file object.

The posixfile object defines the following additional methods:

posixfile.lock(fmt[, len[, start[, whence]]])
Lock the specified section of the file that the file object is referring to. The format is explained below in a table. The len argument specifies the length of the section that should be locked. The default is 0. start specifies the starting offset of the section, where the default is 0. The whence argument

specifies where the offset is relative to. It accepts one of the constants SEEK_SET, SEEK_CUR or SEEK_END. The default is SEEK_SET. For more information about the arguments refer to the fcntl(2) manual page on your system.

posixfile.flags([flags])
Set the specified flags for the file that the file object is referring to. The new flags are ORed with the old flags, unless specified otherwise. The format is explained below in a table. Without the flags argument a string indicating the current flags is returned (this is the same as the ? modifier). For more information about the flags refer to the fcntl(2) manual page on your system.

posixfile.dup()
Duplicate the file object and the underlying file pointer and file descriptor. The resulting object behaves as if it were newly opened.

posixfile.dup2(fd)
Duplicate the file object and the underlying file pointer and file descriptor. The new object will have the given file descriptor. Otherwise the resulting object behaves as if it were newly opened.

posixfile.file()
Return the standard file object that the posixfile object is based on. This is sometimes necessary for functions that insist on a standard file object.

All methods raise IOError when the request fails.

Format characters for the lock() method have the following meaning:

Format	Meaning
u	unlock the specified region
r	request a read lock for the specified section
w	request a write lock for the specified section

In addition the following modifiers can be added to the format:

Modifier	Meaning	Notes
|	wait until the lock has been granted	
?	return the first lock conflicting with the requested lock, or None if there is no conflict.	(1)

Note:

(1) The lock returned is in the format (mode, len, start, whence, pid) where mode is a character representing the type of lock ('r' or 'w'). This modifier prevents a request from being granted; it is for query purposes only.

Format characters for the flags() method have the following meanings:

Format	Meaning
a	append only flag
c	close on exec flag
n	no delay flag (also called non-blocking flag)
s	synchronization flag

In addition the following modifiers can be added to the format:

Modifier	Meaning	Notes
!	turn the specified flags 'off', instead of the default 'on'	(1)
=	replace the flags, instead of the default 'OR' operation	(1)
?	return a string in which the characters represent the flags that are set.	(2)

Notes:

(1) The ! and = modifiers are mutually exclusive.

(2) This string represents the flags after they may have been altered by the same call.

Examples:

```
import posixfile

file = posixfile.open('testfile', 'w')
file.lock('w|')
...
file.lock('u')
file.close()
```

36.13 resource — Resource usage information

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

A single exception is defined for errors:

exception resource.error
: The functions described below may raise this error if the underlying system call failures unexpectedly.

36.13.1 Resource Limits

Resources usage can be limited using the setrlimit() function described below. Each resource is controlled by a pair of limits: a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the getrlimit(2) man page. The resources listed below are supported when the underlying operating system supports them; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

resource.RLIM_INFINITY
: Constant used to represent the limit for an unlimited resource.

resource.getrlimit(resource)
: Returns a tuple (soft, hard) with the current soft and hard limits of resource. Raises ValueError if an invalid resource is specified, or error if the underlying system call fails unexpectedly.

resource.setrlimit(resource, limits)
: Sets new limits of consumption of resource. The limits argument must be a tuple (soft, hard) of two

integers describing the new limits. A value of RLIM_INFINITY can be used to request a limit that is unlimited.

Raises ValueError if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its hard limit. Specifying a limit of RLIM_INFINITY when the hard or system limit for that resource is not unlimited will result in a ValueError. A process with the effective UID of super-user can request any valid limit value, including unlimited, but ValueError will still be raised if the requested limit exceeds the system imposed limit.

setrlimit may also raise error if the underlying system call fails.

These symbols define resources whose consumption can be controlled using the setrlimit() and getrlimit() functions described below. The values of these symbols are exactly the constants used by C programs.

The Unix man page for getrlimit(2) lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences — symbols not defined for a platform will not be available from this module on that platform.

resource.RLIMIT_CORE
The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

resource.RLIMIT_CPU
The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a SIGXCPU signal is sent to the process. (See the signal module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

resource.RLIMIT_FSIZE
The maximum size of a file which the process may create.

resource.RLIMIT_DATA
The maximum size (in bytes) of the process's heap.

resource.RLIMIT_STACK
The maximum size (in bytes) of the call stack for the current process. This only affects the stack of the main thread in a multi-threaded process.

resource.RLIMIT_RSS
The maximum resident set size that should be made available to the process.

resource.RLIMIT_NPROC
The maximum number of processes the current process may create.

resource.RLIMIT_NOFILE
The maximum number of open file descriptors for the current process.

resource.RLIMIT_OFILE
The BSD name for RLIMIT_NOFILE.

resource.RLIMIT_MEMLOCK
The maximum address space which may be locked in memory.

resource.RLIMIT_VMEM
The largest area of mapped memory which the process may occupy.

resource.RLIMIT_AS
The maximum area (in bytes) of address space which may be taken by the process.

36.13.2 Resource Usage

These functions are used to retrieve resource usage information:

resource.getrusage(who)
This function returns an object that describes the resources consumed by either the current process or its children, as specified by the who parameter. The who parameter should be specified using one of the RUSAGE_* constants described below.

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields ru_utime and ru_stime of the return value are floating point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the getrusage(2) man page for detailed information about these values. A brief summary is presented here:

Index	Field	Resource
0	ru_utime	time in user mode (float)
1	ru_stime	time in system mode (float)
2	ru_maxrss	maximum resident set size
3	ru_ixrss	shared memory size
4	ru_idrss	unshared memory size
5	ru_isrss	unshared stack size
6	ru_minflt	page faults not requiring I/O
7	ru_majflt	page faults requiring I/O
8	ru_nswap	number of swap outs
9	ru_inblock	block input operations
10	ru_oublock	block output operations
11	ru_msgsnd	messages sent
12	ru_msgrcv	messages received
13	ru_nsignals	signals received
14	ru_nvcsw	voluntary context switches
15	ru_nivcsw	involuntary context switches

This function will raise a ValueError if an invalid who parameter is specified. It may also raise error exception in unusual circumstances.

Changed in version 2.3: Added access to values as attributes of the returned object.

resource.getpagesize()
Returns the number of bytes in a system page. (This need not be the same as the hardware page size.)

The following RUSAGE_* symbols are passed to the getrusage() function to specify which processes information should be provided for.

resource.RUSAGE_SELF
RUSAGE_SELF should be used to request information pertaining only to the process itself.

resource.RUSAGE_CHILDREN
Pass to getrusage() to request resource information for child processes of the calling process.

resource.RUSAGE_BOTH
Pass to getrusage() to request resources consumed by both the current process and child processes. May not be available on all systems.

36.14 nis — Interface to Sun's NIS (Yellow Pages)

The nis module gives a thin wrapper around the NIS library, useful for central administration of several hosts.

Because NIS exists only on Unix systems, this module is only available for Unix.

The nis module defines the following functions:

nis.match(key, mapname[, domain=default_domain])
: Return the match for key in map mapname, or raise an error (nis.error) if there is none. Both should be strings, key is 8-bit clean. Return value is an arbitrary array of bytes (may contain NULL and other joys).

  Note that mapname is first checked if it is an alias to another name.

  Changed in version 2.5: The domain argument allows overriding the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.

nis.cat(mapname[, domain=default_domain])
: Return a dictionary mapping key to value such that match(key, mapname)==value. Note that both keys and values of the dictionary are arbitrary arrays of bytes.

  Note that mapname is first checked if it is an alias to another name.

  Changed in version 2.5: The domain argument allows overriding the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.

nis.maps([domain=default_domain])
: Return a list of all valid maps.

  Changed in version 2.5: The domain argument allows overriding the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.

nis.get_default_domain()
: Return the system default NIS domain.

  New in version 2.5.

The nis module defines the following exception:

exception nis.error
: An error raised when a NIS function returns an error code.

36.15 syslog — Unix syslog library routines

This module provides an interface to the Unix syslog library routines. Refer to the Unix manual pages for a detailed description of the syslog facility.

This module wraps the system syslog family of routines. A pure Python library that can speak to a syslog server is available in the logging.handlers module as SysLogHandler.

The module defines the following functions:

syslog.syslog(message)
syslog.syslog(priority, message)
: Send the string message to the system logger. A trailing newline is added if necessary. Each message is tagged with a priority composed of a facility and a level. The optional priority argument, which defaults to LOG_INFO, determines the message priority. If the facility is not encoded in priority using logical-or (LOG_INFO | LOG_USER), the value given in the openlog() call is used.

If openlog() has not been called prior to the call to syslog(), openlog() will be called with no arguments.

syslog.openlog([ident[, logoption[, facility]]])
Logging options of subsequent syslog() calls can be set by calling openlog(). syslog() will call openlog() with no arguments if the log is not currently open.

The optional ident keyword argument is a string which is prepended to every message, and defaults to sys.argv[0] with leading path components stripped. The optional logoption keyword argument (default is 0) is a bit field – see below for possible values to combine. The optional facility keyword argument (default is LOG_USER) sets the default facility for messages which do not have a facility explicitly encoded.

syslog.closelog()
Reset the syslog module values and call the system library closelog().

This causes the module to behave as it does when initially imported. For example, openlog() will be called on the first syslog() call (if openlog() hasn't already been called), and ident and other openlog() parameters are reset to defaults.

syslog.setlogmask(maskpri)
Set the priority mask to maskpri and return the previous mask value. Calls to syslog() with a priority level not set in maskpri are ignored. The default is to log all priorities. The function LOG_MASK(pri) calculates the mask for the individual priority pri. The function LOG_UPTO(pri) calculates the mask for all priorities up to and including pri.

The module defines the following constants:

Priority levels (high to low): LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_ERR, LOG_WARNING, LOG_NOTICE, LOG_INFO, LOG_DEBUG.

Facilities: LOG_KERN, LOG_USER, LOG_MAIL, LOG_DAEMON, LOG_AUTH, LOG_LPR, LOG_NEWS, LOG_UUCP, LOG_CRON, LOG_SYSLOG and LOG_LOCAL0 to LOG_LOCAL7.

Log options: LOG_PID, LOG_CONS, LOG_NDELAY, LOG_NOWAIT and LOG_PERROR if defined in <syslog.h>.

36.15.1 Examples

Simple example

A simple set of examples:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

An example of setting some log options, these would include the process ID in logged messages, and write the messages to the destination facility used for mail logging:

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

36.16 commands — Utilities for running commands

Deprecated since version 2.6: The commands module has been removed in Python 3. Use the subprocess module instead.

The commands module contains wrapper functions for os.popen() which take a system command as a string and return any output generated by the command and, optionally, the exit status.

The subprocess module provides more powerful facilities for spawning new processes and retrieving their results. Using the subprocess module is preferable to using the commands module.

Note: In Python 3.x, getstatus() and two undocumented functions (mk2arg() and mkarg()) have been removed. Also, getstatusoutput() and getoutput() have been moved to the subprocess module.

The commands module defines the following functions:

commands.getstatusoutput(cmd)
: Execute the string cmd in a shell with os.popen() and return a 2-tuple (status, output). cmd is actually run as { cmd ; } 2>&1, so that the returned output will contain output or error messages. A trailing newline is stripped from the output. The exit status for the command can be interpreted according to the rules for the C function wait().

commands.getoutput(cmd)
: Like getstatusoutput(), except the exit status is ignored and the return value is a string containing the command's output.

commands.getstatus(file)
: Return the output of ls -ld file as a string. This function uses the getoutput() function, and properly escapes backslashes and dollar signs in the argument.

  Deprecated since version 2.6: This function is nonobvious and useless. The name is also misleading in the presence of getstatusoutput().

Example:

```
>>> import commands
>>> commands.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> commands.getstatusoutput('cat /bin/junk')
(256, 'cat: /bin/junk: No such file or directory')
>>> commands.getstatusoutput('/bin/junk')
(256, 'sh: /bin/junk: not found')
>>> commands.getoutput('ls /bin/ls')
'/bin/ls'
>>> commands.getstatus('/bin/ls')
'-rwxr-xr-x  1 root       13352 Oct 14  1994 /bin/ls'
```

See also:

Module subprocess Module for spawning and managing subprocesses.

CHAPTER

THIRTYSEVEN

MAC OS X SPECIFIC SERVICES

This chapter describes modules that are only available on the Mac OS X platform.

See the chapters MacPython OSA Modules and Undocumented Mac OS modules for more modules, and the HOWTO using-on-mac for a general introduction to Mac-specific Python programming.

Note: Most of the OS X APIs that these modules use are deprecated or removed in recent versions of OS X. Many are not available when Python is executing in 64-bit mode. These modules have been removed in Python 3. You should avoid using them in Python 2.

37.1 ic — Access to the Mac OS X Internet Config

This module provides access to various internet-related preferences set through System Preferences or the Finder.

Note: This module has been removed in Python 3.x.

There is a low-level companion module icglue which provides the basic Internet Config access functionality. This low-level module is not documented, but the docstrings of the routines document the parameters and the routine names are the same as for the Pascal or C API to Internet Config, so the standard IC programmers' documentation can be used if this module is needed.

The ic module defines the error exception and symbolic names for all error codes Internet Config can produce; see the source for details.

exception ic.error
: Exception raised on errors in the ic module.

The ic module defines the following class and function:

class ic.IC([signature[, ic]])
: Create an Internet Config object. The signature is a 4-character creator code of the current application (default `'Pyth'`) which may influence some of ICs settings. The optional ic argument is a low-level icglue.icinstance created beforehand, this may be useful if you want to get preferences from a different config file, etc.

ic.launchurl(url[, hint])

ic.parseurl(data[, start[, end[, hint]]])

ic.mapfile(file)

ic.maptypecreator(type, creator[, filename])

ic.settypecreator(file)
: These functions are "shortcuts" to the methods of the same name, described below.

37.1.1 IC Objects

IC objects have a mapping interface, hence to obtain the mail address you simply get ic['MailAddress']. Assignment also works, and changes the option in the configuration file.

The module knows about various datatypes, and converts the internal IC representation to a "logical" Python data structure. Running the ic module standalone will run a test program that lists all keys and values in your IC database, this will have to serve as documentation.

If the module does not know how to represent the data it returns an instance of the ICOpaqueData type, with the raw data in its data attribute. Objects of this type are also acceptable values for assignment.

Besides the dictionary interface, IC objects have the following methods:

IC.launchurl(url[, hint])
: Parse the given URL, launch the correct application and pass it the URL. The optional hint can be a scheme name such as 'mailto:', in which case incomplete URLs are completed with this scheme. If hint is not provided, incomplete URLs are invalid.

IC.parseurl(data[, start[, end[, hint]]])
: Find a URL somewhere in data and return start position, end position and the URL. The optional start and end can be used to limit the search, so for instance if a user clicks in a long text field you can pass the whole text field and the click-position in start and this routine will return the whole URL in which the user clicked. As above, hint is an optional scheme used to complete incomplete URLs.

IC.mapfile(file)
: Return the mapping entry for the given file, which can be passed as either a filename or an FSSpec() result, and which need not exist.

  The mapping entry is returned as a tuple (version, type, creator, postcreator, flags, extension, appname, postappname, mimetype, entryname), where version is the entry version number, type is the 4-character filetype, creator is the 4-character creator type, postcreator is the 4-character creator code of an optional application to post-process the file after downloading, flags are various bits specifying whether to transfer in binary or ascii and such, extension is the filename extension for this file type, appname is the printable name of the application to which this file belongs, postappname is the name of the postprocessing application, mimetype is the MIME type of this file and entryname is the name of this entry.

IC.maptypecreator(type, creator[, filename])
: Return the mapping entry for files with given 4-character type and creator codes. The optional filename may be specified to further help finding the correct entry (if the creator code is '????', for instance).

  The mapping entry is returned in the same format as for mapfile.

IC.settypecreator(file)
: Given an existing file, specified either as a filename or as an FSSpec() result, set its creator and type correctly based on its extension. The finder is told about the change, so the finder icon will be updated quickly.

37.2 MacOS — Access to Mac OS interpreter features

This module provides access to MacOS specific functionality in the Python interpreter, such as how the interpreter eventloop functions and the like. Use with care.

Note: This module has been removed in Python 3.x.

Note the capitalization of the module name; this is a historical artifact.

MacOS.runtimemodel
: Always `'macho'`, from Python 2.4 on. In earlier versions of Python the value could also be `'ppc'` for the classic Mac OS 8 runtime model or `'carbon'` for the Mac OS 9 runtime model.

MacOS.linkmodel
: The way the interpreter has been linked. As extension modules may be incompatible between linking models, packages could use this information to give more decent error messages. The value is one of `'static'` for a statically linked Python, `'framework'` for Python in a Mac OS X framework, `'shared'` for Python in a standard Unix shared library. Older Pythons could also have the value `'cfm'` for Mac OS 9-compatible Python.

exception MacOS.Error
: This exception is raised on MacOS generated errors, either from functions in this module or from other mac-specific modules like the toolbox interfaces. The arguments are the integer error code (the OSErr value) and a textual description of the error code. Symbolic names for all known error codes are defined in the standard module macerrors.

MacOS.GetErrorString(errno)
: Return the textual description of MacOS error code errno.

MacOS.DebugStr(message[, object])
: On Mac OS X the string is simply printed to stderr (on older Mac OS systems more elaborate functionality was available), but it provides a convenient location to attach a breakpoint in a low-level debugger like gdb.

  Note: Not available in 64-bit mode.

MacOS.SysBeep()
: Ring the bell.

  Note: Not available in 64-bit mode.

MacOS.GetTicks()
: Get the number of clock ticks (1/60th of a second) since system boot.

MacOS.GetCreatorAndType(file)
: Return the file creator and file type as two four-character strings. The file parameter can be a pathname or an FSSpec or FSRef object.

  Note: It is not possible to use an FSSpec in 64-bit mode.

MacOS.SetCreatorAndType(file, creator, type)
: Set the file creator and file type. The file parameter can be a pathname or an FSSpec or FSRef object. creator and type must be four character strings.

  Note: It is not possible to use an FSSpec in 64-bit mode.

MacOS.openrf(name[, mode])
: Open the resource fork of a file. Arguments are the same as for the built-in function open(). The object returned has file-like semantics, but it is not a Python file object, so there may be subtle differences.

MacOS.WMAvailable()
: Checks whether the current process has access to the window manager. The method will return False if the window manager is not available, for instance when running on Mac OS X Server or when logged in via ssh, or when the current interpreter is not running from a fullblown application bundle. A script runs from an application bundle either when it has been started with pythonw instead of python or when running as an applet.

MacOS.splash([resourceid])
: Opens a splash screen by resource id. Use resourceid 0 to close the splash screen.

Note: Not available in 64-bit mode.

37.3 macostools — Convenience routines for file manipulation

This module contains some convenience routines for file-manipulation on the Macintosh. All file parameters can be specified as pathnames, FSRef or FSSpec objects. This module expects a filesystem which supports forked files, so it should not be used on UFS partitions.

Note: This module has been removed in Python 3.

The macostools module defines the following functions:

macostools.copy(src, dst[, createpath[, copytimes]])
: Copy file src to dst. If createpath is non-zero the folders leading to dst are created if necessary. The method copies data and resource fork and some finder information (creator, type, flags) and optionally the creation, modification and backup times (default is to copy them). Custom icons, comments and icon position are not copied.

Note: This function does not work in 64-bit code because it uses APIs that are not available in 64-bit mode.

---

macostools.copytree(src, dst)
: Recursively copy a file tree from src to dst, creating folders as needed. src and dst should be specified as pathnames.

Note: This function does not work in 64-bit code because it uses APIs that are not available in 64-bit mode.

---

macostools.mkalias(src, dst)
: Create a finder alias dst pointing to src.

Note: This function does not work in 64-bit code because it uses APIs that are not available in 64-bit mode.

---

macostools.touched(dst)
Tell the finder that some bits of finder-information such as creator or type for file dst has changed. The file can be specified by pathname or fsspec. This call should tell the finder to redraw the files icon.

Deprecated since version 2.6: The function is a no-op on OS X.

macostools.BUFSIZ
The buffer size for copy, default 1 megabyte.

Note that the process of creating finder aliases is not specified in the Apple documentation. Hence, aliases created with mkalias() could conceivably have incompatible behaviour in some cases.

37.4 findertools — The finder's Apple Events interface

This module contains routines that give Python programs access to some functionality provided by the finder. They are implemented as wrappers around the AppleEvent interface to the finder.

All file and folder parameters can be specified either as full pathnames, or as FSRef or FSSpec objects.

The findertools module defines the following functions:

findertools.launch(file)
Tell the finder to launch file. What launching means depends on the file: applications are started, folders are opened and documents are opened in the correct application.

findertools.Print(file)
Tell the finder to print a file. The behaviour is identical to selecting the file and using the print command in the finder's file menu.

findertools.copy(file, destdir)
Tell the finder to copy a file or folder file to folder destdir. The function returns an Alias object pointing to the new file.

findertools.move(file, destdir)
Tell the finder to move a file or folder file to folder destdir. The function returns an Alias object pointing to the new file.

findertools.sleep()
Tell the finder to put the Macintosh to sleep, if your machine supports it.

findertools.restart()
Tell the finder to perform an orderly restart of the machine.

findertools.shutdown()
Tell the finder to perform an orderly shutdown of the machine.

37.5 EasyDialogs — Basic Macintosh dialogs

The EasyDialogs module contains some simple dialogs for the Macintosh. The dialogs get launched in a separate application which appears in the dock and must be clicked on for the dialogs be displayed. All routines take an optional resource ID parameter id with which one can override the DLOG resource used for the dialog, provided that the dialog items correspond (both type and item number) to those in the default DLOG resource. See source code for details.

Note: This module has been removed in Python 3.x.

The EasyDialogs module defines the following functions:

EasyDialogs.Message(str[, id[, ok]])
Displays a modal dialog with the message text str, which should be at most 255 characters long. The button text defaults to "OK", but is set to the string argument ok if the latter is supplied. Control is returned when the user clicks the "OK" button.

EasyDialogs.AskString(prompt[, default[, id[, ok[, cancel]]]])
Asks the user to input a string value via a modal dialog. prompt is the prompt message, and the optional default supplies the initial value for the string (otherwise "" is used). The text of the "OK" and "Cancel" buttons can be changed with the ok and cancel arguments. All strings can be at most 255 bytes long. AskString() returns the string entered or None in case the user cancelled.

EasyDialogs.AskPassword(prompt[, default[, id[, ok[, cancel]]]])
Asks the user to input a string value via a modal dialog. Like AskString(), but with the text shown as bullets. The arguments have the same meaning as for AskString().

EasyDialogs.AskYesNoCancel(question[, default[, yes[, no[, cancel[, id]]]]])
Presents a dialog with prompt question and three buttons labelled "Yes", "No", and "Cancel". Returns 1 for "Yes", 0 for "No" and -1 for "Cancel". The value of default (or 0 if default is not supplied) is returned when the RETURN key is pressed. The text of the buttons can be changed with the yes, no, and cancel arguments; to prevent a button from appearing, supply "" for the corresponding argument.

EasyDialogs.ProgressBar([title[, maxval[, label[, id]]]])
Displays a modeless progress-bar dialog. This is the constructor for the ProgressBar class described below. title is the text string displayed (default "Working..."), maxval is the value at which progress is complete (default 0, indicating that an indeterminate amount of work remains to be done), and label is the text that is displayed above the progress bar itself.

EasyDialogs.GetArgv([optionlist[commandlist[, addoldfile[, addnewfile[, addfolder[, id]]]]]])
Displays a dialog which aids the user in constructing a command-line argument list. Returns the list in sys.argv format, suitable for passing as an argument to getopt.getopt(). addoldfile, addnewfile, and addfolder are boolean arguments. When nonzero, they enable the user to insert into the command line paths to an existing file, a (possibly) not-yet-existent file, and a folder, respectively. (Note: Option arguments must appear in the command line before file and folder arguments in order to be recognized by getopt.getopt().) Arguments containing spaces can be specified by enclosing them within single or double quotes. A SystemExit exception is raised if the user presses the "Cancel" button.

optionlist is a list that determines a popup menu from which the allowed options are selected. Its items can take one of two forms: optstr or (optstr, descr). When present, descr is a short descriptive string that is displayed in the dialog while this option is selected in the popup menu. The correspondence between optstrs and command-line arguments is:

optstr format	Command-line format
x	-x (short option)
x: or x=	-x (short option with value)
xyz	--xyz (long option)
xyz: or xyz=	--xyz (long option with value)

commandlist is a list of items of the form cmdstr or (cmdstr, descr), where descr is as above. The cmdstrs will appear in a popup menu. When chosen, the text of cmdstr will be appended to the command line as is, except that a trailing `':'` or `'='` (if present) will be trimmed off.

New in version 2.0.

EasyDialogs.AskFileForOpen([message] [, typeList] [, defaultLocation] [, defaultOptionFlags] [, location] [, clientName] [, windowTitle] [, actionButtonLabel] [, cancelButtonLabel] [, preferenceKey] [, popupExtension] [, eventProc] [, previewProc] [, filterProc] [, wanted])

Post a dialog asking the user for a file to open, and return the file selected or None if the user cancelled. message is a text message to display, typeList is a list of 4-char filetypes allowable, defaultLocation is the pathname, FSSpec or FSRef of the folder to show initially, location is the (x, y) position on the screen where the dialog is shown, actionButtonLabel is a string to show instead of "Open" in the OK button, cancelButtonLabel is a string to show instead of "Cancel" in the cancel button, wanted is the type of value wanted as a return: str, unicode, FSSpec, FSRef and subtypes thereof are acceptable.

For a description of the other arguments please see the Apple Navigation Services documentation and the EasyDialogs source code.

EasyDialogs.AskFileForSave([message] [, savedFileName] [, defaultLocation] [, defaultOptionFlags] [, location] [, clientName] [, windowTitle] [, actionButtonLabel] [, cancelButtonLabel] [, preferenceKey] [, popupExtension] [, fileType] [, fileCreator] [, eventProc] [, wanted])

Post a dialog asking the user for a file to save to, and return the file selected or None if the user cancelled. savedFileName is the default for the file name to save to (the return value). See AskFileForOpen() for a description of the other arguments.

EasyDialogs.AskFolder([message] [, defaultLocation] [, defaultOptionFlags] [, location] [, clientName] [, windowTitle] [, actionButtonLabel] [, cancelButtonLabel] [, preferenceKey] [, popupExtension] [, eventProc] [, filterProc] [, wanted])

Post a dialog asking the user to select a folder, and return the folder selected or None if the user cancelled. See AskFileForOpen() for a description of the arguments.

See also:

Navigation Services Reference Programmer's reference documentation for the Navigation Services, a part of the Carbon framework.

37.5.1 ProgressBar Objects

ProgressBar objects provide support for modeless progress-bar dialogs. Both determinate (thermometer style) and indeterminate (barber-pole style) progress bars are supported. The bar will be determinate if its maximum value is greater than zero; otherwise it will be indeterminate.

Changed in version 2.2: Support for indeterminate-style progress bars was added.

The dialog is displayed immediately after creation. If the dialog's "Cancel" button is pressed, or if Cmd-. or ESC is typed, the dialog window is hidden and KeyboardInterrupt is raised (but note that this response does not occur until the progress bar is next updated, typically via a call to inc() or set()). Otherwise, the bar remains visible until the ProgressBar object is discarded.

ProgressBar objects possess the following attributes and methods:

ProgressBar.curval

The current value (of type integer or long integer) of the progress bar. The normal access methods coerce curval between 0 and maxval. This attribute should not be altered directly.

ProgressBar.maxval

The maximum value (of type integer or long integer) of the progress bar; the progress bar (thermometer style) is full when curval equals maxval. If maxval is 0, the bar will be indeterminate (barber-pole). This attribute should not be altered directly.

ProgressBar.title([newstr])

Sets the text in the title bar of the progress dialog to newstr.

ProgressBar.label([newstr])
: Sets the text in the progress box of the progress dialog to newstr.

ProgressBar.set(value[, max])
: Sets the progress bar's curval to value, and also maxval to max if the latter is provided. value is first coerced between 0 and maxval. The thermometer bar is updated to reflect the changes, including a change from indeterminate to determinate or vice versa.

ProgressBar.inc([n])
: Increments the progress bar's curval by n, or by 1 if n is not provided. (Note that n may be negative, in which case the effect is a decrement.) The progress bar is updated to reflect the change. If the bar is indeterminate, this causes one "spin" of the barber pole. The resulting curval is coerced between 0 and maxval if incrementing causes it to fall outside this range.

37.6 FrameWork — Interactive application framework

The FrameWork module contains classes that together provide a framework for an interactive Macintosh application. The programmer builds an application by creating subclasses that override various methods of the bases classes, thereby implementing the functionality wanted. Overriding functionality can often be done on various different levels, i.e. to handle clicks in a single dialog window in a non-standard way it is not necessary to override the complete event handling.

Note: This module has been removed in Python 3.x.

Work on the FrameWork has pretty much stopped, now that PyObjC is available for full Cocoa access from Python, and the documentation describes only the most important functionality, and not in the most logical manner at that. Examine the source or the examples for more details. The following are some comments posted on the MacPython newsgroup about the strengths and limitations of FrameWork:

> The strong point of FrameWork is that it allows you to break into the control-flow at many different places. W, for instance, uses a different way to enable/disable menus and that plugs right in leaving the rest intact. The weak points of FrameWork are that it has no abstract command interface (but that shouldn't be difficult), that its dialog support is minimal and that its control/toolbar support is non-existent.

The FrameWork module defines the following functions:

FrameWork.Application()
: An object representing the complete application. See below for a description of the methods. The default __init__() routine creates an empty window dictionary and a menu bar with an apple menu.

FrameWork.MenuBar()
: An object representing the menubar. This object is usually not created by the user.

FrameWork.Menu(bar, title[, after])
: An object representing a menu. Upon creation you pass the MenuBar the menu appears in, the title string and a position (1-based) after where the menu should appear (default: at the end).

FrameWork.MenuItem(menu, title[, shortcut, callback])
: Create a menu item object. The arguments are the menu to create, the item title string and optionally the keyboard shortcut and a callback routine. The callback is called with the arguments menu-id, item number within menu (1-based), current front window and the event record.

  Instead of a callable object the callback can also be a string. In this case menu selection causes the lookup of a method in the topmost window and the application. The method name is the callback

string with 'domenu_' prepended.

Calling the MenuBar fixmenudimstate() method sets the correct dimming for all menu items based on the current front window.

FrameWork.Separator(menu)
Add a separator to the end of a menu.

FrameWork.SubMenu(menu, label)
Create a submenu named label under menu menu. The menu object is returned.

FrameWork.Window(parent)
Creates a (modeless) window. Parent is the application object to which the window belongs. The window is not displayed until later.

FrameWork.DialogWindow(parent)
Creates a modeless dialog window.

FrameWork.windowbounds(width, height)
Return a (left, top, right, bottom) tuple suitable for creation of a window of given width and height. The window will be staggered with respect to previous windows, and an attempt is made to keep the whole window on-screen. However, the window will however always be the exact size given, so parts may be offscreen.

FrameWork.setwatchcursor()
Set the mouse cursor to a watch.

FrameWork.setarrowcursor()
Set the mouse cursor to an arrow.

37.6.1 Application Objects

Application objects have the following methods, among others:

Application.makeusermenus()
Override this method if you need menus in your application. Append the menus to the attribute menubar.

Application.getabouttext()
Override this method to return a text string describing your application. Alternatively, override the do_about() method for more elaborate "about" messages.

Application.mainloop([mask[, wait]])
This routine is the main event loop, call it to set your application rolling. Mask is the mask of events you want to handle, wait is the number of ticks you want to leave to other concurrent application (default 0, which is probably not a good idea). While raising self to exit the mainloop is still supported it is not recommended: call self._quit() instead.

The event loop is split into many small parts, each of which can be overridden. The default methods take care of dispatching events to windows and dialogs, handling drags and resizes, Apple Events, events for non-FrameWork windows, etc.

In general, all event handlers should return 1 if the event is fully handled and 0 otherwise (because the front window was not a FrameWork window, for instance). This is needed so that update events and such can be passed on to other windows like the Sioux console window. Calling MacOS.HandleEvent() is not allowed within our_dispatch or its callees, since this may result in an infinite loop if the code is called through the Python inner-loop event handler.

Application.asyncevents(onoff)
Call this method with a nonzero parameter to enable asynchronous event handling. This will tell

the inner interpreter loop to call the application event handler async_dispatch whenever events are available. This will cause FrameWork window updates and the user interface to remain working during long computations, but will slow the interpreter down and may cause surprising results in non-reentrant code (such as FrameWork itself). By default async_dispatch will immediately call our_dispatch but you may override this to handle only certain events asynchronously. Events you do not handle will be passed to Sioux and such.

The old on/off value is returned.

Application._quit()
Terminate the running mainloop() call at the next convenient moment.

Application.do_char(c, event)
The user typed character c. The complete details of the event can be found in the event structure. This method can also be provided in a Window object, which overrides the application-wide handler if the window is frontmost.

Application.do_dialogevent(event)
Called early in the event loop to handle modeless dialog events. The default method simply dispatches the event to the relevant dialog (not through the DialogWindow object involved). Override if you need special handling of dialog events (keyboard shortcuts, etc).

Application.idle(event)
Called by the main event loop when no events are available. The null-event is passed (so you can look at mouse position, etc).

37.6.2 Window Objects

Window objects have the following methods, among others:

Window.open()
Override this method to open a window. Store the Mac OS window-id in self.wid and call the do_postopen() method to register the window with the parent application.

Window.close()
Override this method to do any special processing on window close. Call the do_postclose() method to cleanup the parent state.

Window.do_postresize(width, height, macoswindowid)
Called after the window is resized. Override if more needs to be done than calling InvalRect.

Window.do_contentclick(local, modifiers, event)
The user clicked in the content part of a window. The arguments are the coordinates (window-relative), the key modifiers and the raw event.

Window.do_update(macoswindowid, event)
An update event for the window was received. Redraw the window.

Window.do_activate(activate, event)
The window was activated (activate == 1) or deactivated (activate == 0). Handle things like focus highlighting, etc.

37.6.3 ControlsWindow Object

ControlsWindow objects have the following methods besides those of Window objects:

ControlsWindow.do_controlhit(window, control, pcode, event)
Part pcode of control control was hit by the user. Tracking and such has already been taken care of.

37.6.4 ScrolledWindow Object

ScrolledWindow objects are ControlsWindow objects with the following extra methods:

ScrolledWindow.scrollbars([wantx[, wanty]])
: Create (or destroy) horizontal and vertical scrollbars. The arguments specify which you want (default: both). The scrollbars always have minimum 0 and maximum 32767.

ScrolledWindow.getscrollbarvalues()
: You must supply this method. It should return a tuple (x, y) giving the current position of the scrollbars (between 0 and 32767). You can return None for either to indicate the whole document is visible in that direction.

ScrolledWindow.updatescrollbars()
: Call this method when the document has changed. It will call getscrollbarvalues() and update the scrollbars.

ScrolledWindow.scrollbar_callback(which, what, value)
: Supplied by you and called after user interaction. which will be `'x'` or `'y'`, what will be `'-'`, `'--'`, `'set'`, `'++'` or `'+'`. For `'set'`, value will contain the new scrollbar position.

ScrolledWindow.scalebarvalues(absmin, absmax, curmin, curmax)
: Auxiliary method to help you calculate values to return from getscrollbarvalues(). You pass document minimum and maximum value and topmost (leftmost) and bottommost (rightmost) visible values and it returns the correct number or None.

ScrolledWindow.do_activate(onoff, event)
: Takes care of dimming/highlighting scrollbars when a window becomes frontmost. If you override this method, call this one at the end of your method.

ScrolledWindow.do_postresize(width, height, window)
: Moves scrollbars to the correct position. Call this method initially if you override it.

ScrolledWindow.do_controlhit(window, control, pcode, event)
: Handles scrollbar interaction. If you override it call this method first, a nonzero return value indicates the hit was in the scrollbars and has been handled.

37.6.5 DialogWindow Objects

DialogWindow objects have the following methods besides those of Window objects:

DialogWindow.open(resid)
: Create the dialog window, from the DLOG resource with id resid. The dialog object is stored in self.wid.

DialogWindow.do_itemhit(item, event)
: Item number item was hit. You are responsible for redrawing toggle buttons, etc.

37.7 autoGIL — Global Interpreter Lock handling in event loops

The autoGIL module provides a function installAutoGIL() that automatically locks and unlocks Python's Global Interpreter Lock when running an event loop.

Note: This module has been removed in Python 3.x.

---

exception autoGIL.AutoGILError
: Raised if the observer callback cannot be installed, for example because the current thread does not have a run loop.

autoGIL.installAutoGIL()
: Install an observer callback in the event loop (CFRunLoop) for the current thread, that will lock and unlock the Global Interpreter Lock (GIL) at appropriate times, allowing other Python threads to run while the event loop is idle.

  Availability: OSX 10.1 or later.

37.8 Mac OS Toolbox Modules

These are a set of modules that provide interfaces to various legacy Mac OS toolboxes. If applicable the module will define a number of Python objects for the various structures declared by the toolbox, and operations will be implemented as methods of the object. Other operations will be implemented as functions in the module. Not all operations possible in C will also be possible in Python (callbacks are often a problem), and parameters will occasionally be different in Python (input and output buffers, especially). All methods and functions have a __doc__ string describing their arguments and return values, and for additional description you are referred to Inside Macintosh or similar works.

These modules all live in a package called Carbon. Despite that name they are not all part of the Carbon framework: CF is really in the CoreFoundation framework and Qt is in the QuickTime framework. The normal use pattern is

```
from Carbon import AE
```

Note: Most of the OS X APIs that these modules use are deprecated or removed in recent versions of OS X. Many are not available when Python is executing in 64-bit mode. The Carbon modules have been removed in Python 3. You should avoid using them in Python 2.

37.8.1 Carbon.AE — Apple Events

37.8.2 Carbon.AH — Apple Help

37.8.3 Carbon.App — Appearance Manager

37.8.4 Carbon.Appearance — Appearance Manager constants

37.8.5 Carbon.CF — Core Foundation

The CFBase, CFArray, CFData, CFDictionary, CFString and CFURL objects are supported, some only partially.

37.8.6 Carbon.CG — Core Graphics

37.8.7 Carbon.CarbonEvt — Carbon Event Manager

37.8.8 Carbon.CarbonEvents — Carbon Event Manager constants

37.8.9 Carbon.Cm — Component Manager

37.8.10 Carbon.Components — Component Manager constants

37.8.11 Carbon.ControlAccessor — Control Manager accssors

37.8.12 Carbon.Controls — Control Manager constants

37.8.13 Carbon.CoreFounation — CoreFounation constants

37.8.14 Carbon.CoreGraphics — CoreGraphics constants

37.8.15 Carbon.Ctl — Control Manager

37.8.16 Carbon.Dialogs — Dialog Manager constants

37.8.17 Carbon.Dlg — Dialog Manager

37.8.18 Carbon.Drag — Drag and Drop Manager

37.8.19 Carbon.Dragconst — Drag and Drop Manager constants

37.8.20 Carbon.Events — Event Manager constants

37.8.21 Carbon.Evt — Event Manager

37.8.22 Carbon.File — File Manager

37.8.23 Carbon.Files — File Manager constants

37.8.24 Carbon.Fm — Font Manager

37.8.25 Carbon.Folder — Folder Manager

37.8.26 Carbon.Folders — Folder Manager constants

37.8.27 Carbon.Fonts — Font Manager constants

37.8.28 Carbon.Help — Help Manager

37.8.29 Carbon.IBCarbon — Carbon InterfaceBuilder

37.8.30 Carbon.IBCarbonRuntime — Carbon InterfaceBuilder constants

37.8.31 Carbon.Icn — Carbon Icon Manager

37.8.32 Carbon.Icons — Carbon Icon Manager constants

37.8.33 Carbon.Launch — Carbon Launch Services

The Scrap Manager supports the simplest form of cut & paste operations on the Macintosh. It can be use for both inter- and intra-application clipboard operations.

The Scrap module provides low-level access to the functions of the Scrap Manager. It contains the following functions:

Carbon.Scrap.InfoScrap()
: Return current information about the scrap. The information is encoded as a tuple containing the fields (size, handle, count, state, path).

Field	Meaning
size	Size of the scrap in bytes.
handle	Resource object representing the scrap.
count	Serial number of the scrap contents.
state	Integer; positive if in memory, 0 if on disk, negative if uninitialized.
path	Filename of the scrap when stored on disk.

See also:

Scrap Manager Apple's documentation for the Scrap Manager gives a lot of useful information about using the Scrap Manager in applications.

37.8.53 Carbon.Snd — Sound Manager

37.8.54 Carbon.Sound — Sound Manager constants

37.8.55 Carbon.TE — TextEdit

37.8.56 Carbon.TextEdit — TextEdit constants

37.8.57 Carbon.Win — Window Manager

37.8.58 Carbon.Windows — Window Manager constants

37.9 ColorPicker — Color selection dialog

The ColorPicker module provides access to the standard color picker dialog.

Note: This module has been removed in Python 3.x.

ColorPicker.GetColor(prompt, rgb)
: Show a standard color selection dialog and allow the user to select a color. The user is given instruction by the prompt string, and the default color is set to rgb. rgb must be a tuple giving the red, green, and blue components of the color. GetColor() returns a tuple giving the user's selected color and a flag indicating whether they accepted the selection of cancelled.

CHAPTER

THIRTYEIGHT

MACPYTHON OSA MODULES

This chapter describes the current implementation of the Open Scripting Architecture (OSA, also commonly referred to as AppleScript) for Python, allowing you to control scriptable applications from your Python program, and with a fairly pythonic interface. Development on this set of modules has stopped.

For a description of the various components of AppleScript and OSA, and to get an understanding of the architecture and terminology, you should read Apple's documentation. The "Applescript Language Guide" explains the conceptual model and the terminology, and documents the standard suite. The "Open Scripting Architecture" document explains how to use OSA from an application programmers point of view. In the Apple Help Viewer these books are located in the Developer Documentation, Core Technologies section.

As an example of scripting an application, the following piece of AppleScript will get the name of the frontmost Finder window and print it:

```
tell application "Finder"
    get name of window 1
end tell
```

In Python, the following code fragment will do the same:

```
import Finder

f = Finder.Finder()
print f.get(f.window(1).name)
```

As distributed the Python library includes packages that implement the standard suites, plus packages that interface to a small number of common applications.

To send AppleEvents to an application you must first create the Python package interfacing to the terminology of the application (what Script Editor calls the "Dictionary"). This can be done from within the PythonIDE or by running the gensuitemodule.py module as a standalone program from the command line.

The generated output is a package with a number of modules, one for every suite used in the program plus an __init__ module to glue it all together. The Python inheritance graph follows the AppleScript inheritance graph, so if a program's dictionary specifies that it includes support for the Standard Suite, but extends one or two verbs with extra arguments then the output suite will contain a module Standard_Suite that imports and re-exports everything from StdSuites.Standard_Suite but overrides the methods that have extra functionality. The output of gensuitemodule is pretty readable, and contains the documentation that was in the original AppleScript dictionary in Python docstrings, so reading it is a good source of documentation.

The output package implements a main class with the same name as the package which contains all the AppleScript verbs as methods, with the direct object as the first argument and all optional parameters as keyword arguments. AppleScript classes are also implemented as Python classes, as are comparisons and all the other thingies.

The main Python class implementing the verbs also allows access to the properties and elements declared in the AppleScript class "application". In the current release that is as far as the object orientation goes, so in the example above we need to use f.get(f.window(1).name) instead of the more Pythonic f.window(1).name.get().

If an AppleScript identifier is not a Python identifier the name is mangled according to a small number of rules:

- spaces are replaced with underscores
- other non-alphanumeric characters are replaced with _xx_ where xx is the hexadecimal character value
- any Python reserved word gets an underscore appended

Python also has support for creating scriptable applications in Python, but The following modules are relevant to MacPython AppleScript support:

38.1 gensuitemodule — Generate OSA stub packages

The gensuitemodule module creates a Python package implementing stub code for the AppleScript suites that are implemented by a specific application, according to its AppleScript dictionary.

It is usually invoked by the user through the PythonIDE, but it can also be run as a script from the command line (pass --help for help on the options) or imported from Python code. For an example of its use see Mac/scripts/genallsuites.py in a source distribution, which generates the stub packages that are included in the standard library.

It defines the following public functions:

gensuitemodule.is_scriptable(application)
: Returns true if application, which should be passed as a pathname, appears to be scriptable. Take the return value with a grain of salt: Internet Explorer appears not to be scriptable but definitely is.

gensuitemodule.processfile(application[, output, basepkgname, edit_modnames, creatorsignature, dump, verbose])
: Create a stub package for application, which should be passed as a full pathname. For a .app bundle this is the pathname to the bundle, not to the executable inside the bundle; for an unbundled CFM application you pass the filename of the application binary.

  This function asks the application for its OSA terminology resources, decodes these resources and uses the resultant data to create the Python code for the package implementing the client stubs.

  output is the pathname where the resulting package is stored, if not specified a standard "save file as" dialog is presented to the user. basepkgname is the base package on which this package will build, and defaults to StdSuites. Only when generating StdSuites itself do you need to specify this. edit_modnames is a dictionary that can be used to change modulenames that are too ugly after name mangling. creator_signature can be used to override the 4-char creator code, which is normally obtained from the PkgInfo file in the package or from the CFM file creator signature. When dump is given it should refer to a file object, and processfile will stop after decoding the resources and dump the Python representation of the terminology resources to this file. verbose should also be a file object, and specifying it will cause processfile to tell you what it is doing.

gensuitemodule.processfile_fromresource(application[, output, basepkgname, edit_modnames, creatorsignature, dump, verbose])
: This function does the same as processfile, except that it uses a different method to get the terminology resources. It opens application as a resource file and reads all "aete" and "aeut" resources from this file.

38.2 aetools — OSA client support

The aetools module contains the basic functionality on which Python AppleScript client support is built. It also imports and re-exports the core functionality of the aetypes and aepack modules. The stub packages generated by gensuitemodule import the relevant portions of aetools, so usually you do not need to import it yourself. The exception to this is when you cannot use a generated suite package and need lower-level access to scripting.

The aetools module itself uses the AppleEvent support provided by the Carbon.AE module. This has one drawback: you need access to the window manager, see section osx-gui-scripts for details. This restriction may be lifted in future releases.

Note: This module has been removed in Python 3.x.

The aetools module defines the following functions:

aetools.packevent(ae, parameters, attributes)
: Stores parameters and attributes in a pre-created Carbon.AE.AEDesc object. parameters and attributes are dictionaries mapping 4-character OSA parameter keys to Python objects. The objects are packed using aepack.pack().

aetools.unpackevent(ae[, formodulename])
: Recursively unpacks a Carbon.AE.AEDesc event to Python objects. The function returns the parameter dictionary and the attribute dictionary. The formodulename argument is used by generated stub packages to control where AppleScript classes are looked up.

aetools.keysubst(arguments, keydict)
: Converts a Python keyword argument dictionary arguments to the format required by packevent by replacing the keys, which are Python identifiers, by the four-character OSA keys according to the mapping specified in keydict. Used by the generated suite packages.

aetools.enumsubst(arguments, key, edict)
: If the arguments dictionary contains an entry for key convert the value for that entry according to dictionary edict. This converts human-readable Python enumeration names to the OSA 4-character codes. Used by the generated suite packages.

The aetools module defines the following class:

class aetools.TalkTo([signature=None, start=0, timeout=0])
: Base class for the proxy used to talk to an application. signature overrides the class attribute _signature (which is usually set by subclasses) and is the 4-char creator code defining the application to talk to. start can be set to true to enable running the application on class instantiation. timeout can be specified to change the default timeout used while waiting for an AppleEvent reply.

TalkTo._start()
: Test whether the application is running, and attempt to start it if not.

TalkTo.send(code, subcode[, parameters, attributes])
: Create the AppleEvent Carbon.AE.AEDesc for the verb with the OSA designation code, subcode (which are the usual 4-character strings), pack the parameters and attributes into it, send it to the target application, wait for the reply, unpack the reply with unpackevent and return the reply appleevent, the unpacked return values as a dictionary and the return attributes.

38.3 aepack — Conversion between Python variables and AppleEvent data containers

The aepack module defines functions for converting (packing) Python variables to AppleEvent descriptors and back (unpacking). Within Python the AppleEvent descriptor is handled by Python objects of built-in type AEDesc, defined in module Carbon.AE.

Note: This module has been removed in Python 3.x.

The aepack module defines the following functions:

aepack.pack(x[, forcetype])

Returns an AEDesc object containing a conversion of Python value x. If forcetype is provided it specifies the descriptor type of the result. Otherwise, a default mapping of Python types to Apple Event descriptor types is used, as follows:

Python type	descriptor type
FSSpec	typeFSS
FSRef	typeFSRef
Alias	typeAlias
integer	typeLong (32 bit integer)
float	typeFloat (64 bit floating point)
string	typeText
unicode	typeUnicodeText
list	typeAEList
dictionary	typeAERecord
instance	see below

If x is a Python instance then this function attempts to call an __aepack__() method. This method should return an AEDesc object.

If the conversion x is not defined above, this function returns the Python string representation of a value (the repr() function) encoded as a text descriptor.

aepack.unpack(x[, formodulename])

x must be an object of type AEDesc. This function returns a Python object representation of the data in the Apple Event descriptor x. Simple AppleEvent data types (integer, text, float) are returned as their obvious Python counterparts. Apple Event lists are returned as Python lists, and the list elements are recursively unpacked. Object references (ex. line 3 of document 1) are returned as instances of aetypes.ObjectSpecifier, unless formodulename is specified. AppleEvent descriptors with descriptor type typeFSS are returned as FSSpec objects. AppleEvent record descriptors are returned as Python dictionaries, with 4-character string keys and elements recursively unpacked.

The optional formodulename argument is used by the stub packages generated by gensuitemodule, and ensures that the OSA classes for object specifiers are looked up in the correct module. This ensures that if, say, the Finder returns an object specifier for a window you get an instance of Finder.Window and not a generic aetypes.Window. The former knows about all the properties and elements a window has in the Finder, while the latter knows no such things.

See also:

Module Carbon.AE Built-in access to Apple Event Manager routines.

Module aetypes Python definitions of codes for Apple Event descriptor types.

38.4 aetypes — AppleEvent objects

The aetypes defines classes used to represent Apple Event data descriptors and Apple Event object specifiers.

Apple Event data is contained in descriptors, and these descriptors are typed. For many descriptors the Python representation is simply the corresponding Python type: typeText in OSA is a Python string, typeFloat is a float, etc. For OSA types that have no direct Python counterpart this module declares classes. Packing and unpacking instances of these classes is handled automatically by aepack.

An object specifier is essentially an address of an object implemented in an Apple Event server. An Apple Event specifier is used as the direct object for an Apple Event or as the argument of an optional parameter. The aetypes module contains the base classes for OSA classes and properties, which are used by the packages generated by gensuitemodule to populate the classes and properties in a given suite.

For reasons of backward compatibility, and for cases where you need to script an application for which you have not generated the stub package this module also contains object specifiers for a number of common OSA classes such as Document, Window, Character, etc.

Note: This module has been removed in Python 3.x.

The AEObjects module defines the following classes to represent Apple Event descriptor data:

class aetypes.Unknown(type, data)
: The representation of OSA descriptor data for which the aepack and aetypes modules have no support, i.e. anything that is not represented by the other classes here and that is not equivalent to a simple Python value.

class aetypes.Enum(enum)
: An enumeration value with the given 4-character string value.

class aetypes.InsertionLoc(of, pos)
: Position pos in object of.

class aetypes.Boolean(bool)
: A boolean.

class aetypes.StyledText(style, text)
: Text with style information (font, face, etc) included.

class aetypes.AEText(script, style, text)
: Text with script system and style information included.

class aetypes.IntlText(script, language, text)
: Text with script system and language information included.

class aetypes.IntlWritingCode(script, language)
: Script system and language information.

class aetypes.QDPoint(v, h)
: A quickdraw point.

class aetypes.QDRectangle(v0, h0, v1, h1)
: A quickdraw rectangle.

class aetypes.RGBColor(r, g, b)
: A color.

class aetypes.Type(type)
: An OSA type value with the given 4-character name.

class aetypes.Keyword(name)
An OSA keyword with the given 4-character name.

class aetypes.Range(start, stop)
A range.

class aetypes.Ordinal(abso)
Non-numeric absolute positions, such as "firs", first, or "midd", middle.

class aetypes.Logical(logc, term)
The logical expression of applying operator logc to term.

class aetypes.Comparison(obj1, relo, obj2)
The comparison relo of obj1 to obj2.

The following classes are used as base classes by the generated stub packages to represent AppleScript classes and properties in Python:

class aetypes.ComponentItem(which[, fr])
Abstract baseclass for an OSA class. The subclass should set the class attribute want to the 4-character OSA class code. Instances of subclasses of this class are equivalent to AppleScript Object Specifiers. Upon instantiation you should pass a selector in which, and optionally a parent object in fr.

class aetypes.NProperty(fr)
Abstract baseclass for an OSA property. The subclass should set the class attributes want and which to designate which property we are talking about. Instances of subclasses of this class are Object Specifiers.

class aetypes.ObjectSpecifier(want, form, seld[, fr])
Base class of ComponentItem and NProperty, a general OSA Object Specifier. See the Apple Open Scripting Architecture documentation for the parameters. Note that this class is not abstract.

38.5 MiniAEFrame — Open Scripting Architecture server support

The module MiniAEFrame provides a framework for an application that can function as an Open Scripting Architecture (OSA) server, i.e. receive and process AppleEvents. It can be used in conjunction with FrameWork or standalone. As an example, it is used in PythonCGISlave.

The MiniAEFrame module defines the following classes:

class MiniAEFrame.AEServer
A class that handles AppleEvent dispatch. Your application should subclass this class together with either MiniApplication or FrameWork.Application. Your __init__() method should call the __init__() method for both classes.

class MiniAEFrame.MiniApplication
A class that is more or less compatible with FrameWork.Application but with less functionality. Its event loop supports the apple menu, command-dot and AppleEvents; other events are passed on to the Python interpreter and/or Sioux. Useful if your application wants to use AEServer but does not provide its own windows, etc.

38.5.1 AEServer Objects

AEServer.installaehandler(classe, type, callback)
Installs an AppleEvent handler. classe and type are the four-character OSA Class and Type designators, '****' wildcards are allowed. When a matching AppleEvent is received the parameters are decoded and your callback is invoked.

AEServer.callback(_object, **kwargs)

Your callback is called with the OSA Direct Object as first positional parameter. The other parameters are passed as keyword arguments, with the 4-character designator as name. Three extra keyword parameters are passed: _class and _type are the Class and Type designators and _attributes is a dictionary with the AppleEvent attributes.

The return value of your method is packed with aetools.packevent() and sent as reply.

Note that there are some serious problems with the current design. AppleEvents which have non-identifier 4-character designators for arguments are not implementable, and it is not possible to return an error to the originator. This will be addressed in a future release.

In addition, support modules have been pre-generated for Finder, Terminal, Explorer, Netscape, CodeWarrior, SystemEvents and StdSuites.

CHAPTER

THIRTYNINE

SGI IRIX SPECIFIC SERVICES

The modules described in this chapter provide interfaces to features that are unique to SGI's IRIX operating system (versions 4 and 5).

39.1 al — Audio functions on the SGI

Deprecated since version 2.6: The al module has been removed in Python 3.

This module provides access to the audio facilities of the SGI Indy and Indigo workstations. See section 3A of the IRIX man pages for details. You'll need to read those man pages to understand what these functions do! Some of the functions are not available in IRIX releases before 4.0.5. Again, see the manual to check whether a specific function is available on your platform.

All functions and methods defined in this module are equivalent to the C functions with AL prefixed to their name.

Symbolic constants from the C header file <audio.h> are defined in the standard module AL, see below.

> Warning: The current version of the audio library may dump core when bad argument values are passed rather than returning an error status. Unfortunately, since the precise circumstances under which this may happen are undocumented and hard to check, the Python interface can provide no protection against this kind of problems. (One example is specifying an excessive queue size — there is no documented upper limit.)

The module defines the following functions:

al.openport(name, direction[, config])
: The name and direction arguments are strings. The optional config argument is a configuration object as returned by newconfig(). The return value is an audio port object; methods of audio port objects are described below.

al.newconfig()
: The return value is a new audio configuration object; methods of audio configuration objects are described below.

al.queryparams(device)
: The device argument is an integer. The return value is a list of integers containing the data returned by ALqueryparams().

al.getparams(device, list)
: The device argument is an integer. The list argument is a list such as returned by queryparams(); it is modified in place (!).

al.setparams(device, list)
: The device argument is an integer. The list argument is a list such as returned by queryparams().

39.1.1 Configuration Objects

Configuration objects returned by newconfig() have the following methods:

audio configuration.getqueuesize()
: Return the queue size.

audio configuration.setqueuesize(size)
: Set the queue size.

audio configuration.getwidth()
: Get the sample width.

audio configuration.setwidth(width)
: Set the sample width.

audio configuration.getchannels()
: Get the channel count.

audio configuration.setchannels(nchannels)
: Set the channel count.

audio configuration.getsampfmt()
: Get the sample format.

audio configuration.setsampfmt(sampfmt)
: Set the sample format.

audio configuration.getfloatmax()
: Get the maximum value for floating sample formats.

audio configuration.setfloatmax(floatmax)
: Set the maximum value for floating sample formats.

39.1.2 Port Objects

Port objects, as returned by openport(), have the following methods:

audio port.closeport()
: Close the port.

audio port.getfd()
: Return the file descriptor as an int.

audio port.getfilled()
: Return the number of filled samples.

audio port.getfillable()
: Return the number of fillable samples.

audio port.readsamps(nsamples)
: Read a number of samples from the queue, blocking if necessary. Return the data as a string containing the raw data, (e.g., 2 bytes per sample in big-endian byte order (high byte, low byte) if you have set the sample width to 2 bytes).

audio port.writesamps(samples)
: Write samples into the queue, blocking if necessary. The samples are encoded as described for the readsamps() return value.

audio port.getfillpoint()
: Return the 'fill point'.

audio port.setfillpoint(fillpoint)
: Set the 'fill point'.

audio port.getconfig()
: Return a configuration object containing the current configuration of the port.

audio port.setconfig(config)
: Set the configuration from the argument, a configuration object.

audio port.getstatus(list)
: Get status information on last error.

39.2 AL — Constants used with the al module

Deprecated since version 2.6: The AL module has been removed in Python 3.

This module defines symbolic constants needed to use the built-in module al (see above); they are equivalent to those defined in the C header file <audio.h> except that the name prefix AL_ is omitted. Read the module source for a complete list of the defined names. Suggested use:

```
import al
from AL import *
```

39.3 cd — CD-ROM access on SGI systems

Deprecated since version 2.6: The cd module has been removed in Python 3.

This module provides an interface to the Silicon Graphics CD library. It is available only on Silicon Graphics systems.

The way the library works is as follows. A program opens the CD-ROM device with open() and creates a parser to parse the data from the CD with createparser(). The object returned by open() can be used to read data from the CD, but also to get status information for the CD-ROM device, and to get information about the CD, such as the table of contents. Data from the CD is passed to the parser, which parses the frames, and calls any callback functions that have previously been added.

An audio CD is divided into tracks or programs (the terms are used interchangeably). Tracks can be subdivided into indices. An audio CD contains a table of contents which gives the starts of the tracks on the CD. Index 0 is usually the pause before the start of a track. The start of the track as given by the table of contents is normally the start of index 1.

Positions on a CD can be represented in two ways. Either a frame number or a tuple of three values, minutes, seconds and frames. Most functions use the latter representation. Positions can be both relative to the beginning of the CD, and to the beginning of the track.

Module cd defines the following functions and constants:

cd.createparser()
: Create and return an opaque parser object. The methods of the parser object are described below.

cd.msftoframe(minutes, seconds, frames)
: Converts a (minutes, seconds, frames) triple representing time in absolute time code into the corresponding CD frame number.

cd.open([device[, mode]])
: Open the CD-ROM device. The return value is an opaque player object; methods of the player object are described below. The device is the name of the SCSI device file, e.g. `'/dev/scsi/sc0d4l0'`, or None. If omitted or None, the hardware inventory is consulted to locate a CD-ROM drive. The mode, if not omitted, should be the string `'r'`.

The module defines the following variables:

exception cd.error
: Exception raised on various errors.

cd.DATASIZE
: The size of one frame's worth of audio data. This is the size of the audio data as passed to the callback of type audio.

cd.BLOCKSIZE
: The size of one uninterpreted frame of audio data.

The following variables are states as returned by getstatus():

cd.READY
: The drive is ready for operation loaded with an audio CD.

cd.NODISC
: The drive does not have a CD loaded.

cd.CDROM
: The drive is loaded with a CD-ROM. Subsequent play or read operations will return I/O errors.

cd.ERROR
: An error occurred while trying to read the disc or its table of contents.

cd.PLAYING
: The drive is in CD player mode playing an audio CD through its audio jacks.

cd.PAUSED
: The drive is in CD layer mode with play paused.

cd.STILL
: The equivalent of PAUSED on older (non 3301) model Toshiba CD-ROM drives. Such drives have never been shipped by SGI.

cd.audio
cd.pnum
cd.index
cd.ptime
cd.atime
cd.catalog
cd.ident
cd.control
: Integer constants describing the various types of parser callbacks that can be set by the addcallback() method of CD parser objects (see below).

39.3.1 Player Objects

Player objects (returned by open()) have the following methods:

CD player.allowremoval()
Unlocks the eject button on the CD-ROM drive permitting the user to eject the caddy if desired.

CD player.bestreadsize()
Returns the best value to use for the num_frames parameter of the readda() method. Best is defined as the value that permits a continuous flow of data from the CD-ROM drive.

CD player.close()
Frees the resources associated with the player object. After calling close(), the methods of the object should no longer be used.

CD player.eject()
Ejects the caddy from the CD-ROM drive.

CD player.getstatus()
Returns information pertaining to the current state of the CD-ROM drive. The returned information is a tuple with the following values: state, track, rtime, atime, ttime, first, last, scsi_audio, cur_block. rtime is the time relative to the start of the current track; atime is the time relative to the beginning of the disc; ttime is the total time on the disc. For more information on the meaning of the values, see the man page CDgetstatus(3dm). The value of state is one of the following: ERROR, NODISC, READY, PLAYING, PAUSED, STILL, or CDROM.

CD player.gettrackinfo(track)
Returns information about the specified track. The returned information is a tuple consisting of two elements, the start time of the track and the duration of the track.

CD player.msftoblock(min, sec, frame)
Converts a minutes, seconds, frames triple representing a time in absolute time code into the corresponding logical block number for the given CD-ROM drive. You should use msftoframe() rather than msftoblock() for comparing times. The logical block number differs from the frame number by an offset required by certain CD-ROM drives.

CD player.play(start, play)
Starts playback of an audio CD in the CD-ROM drive at the specified track. The audio output appears on the CD-ROM drive's headphone and audio jacks (if fitted). Play stops at the end of the disc. start is the number of the track at which to start playing the CD; if play is 0, the CD will be set to an initial paused state. The method togglepause() can then be used to commence play.

CD player.playabs(minutes, seconds, frames, play)
Like play(), except that the start is given in minutes, seconds, and frames instead of a track number.

CD player.playtrack(start, play)
Like play(), except that playing stops at the end of the track.

CD player.playtrackabs(track, minutes, seconds, frames, play)
Like play(), except that playing begins at the specified absolute time and ends at the end of the specified track.

CD player.preventremoval()
Locks the eject button on the CD-ROM drive thus preventing the user from arbitrarily ejecting the caddy.

CD player.readda(num_frames)
Reads the specified number of frames from an audio CD mounted in the CD-ROM drive. The return value is a string representing the audio frames. This string can be passed unaltered to the parseframe() method of the parser object.

CD player.seek(minutes, seconds, frames)
Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM.

The pointer is set to an absolute time code location specified in minutes, seconds, and frames. The return value is the logical block number to which the pointer has been set.

CD player.seekblock(block)
Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified logical block number. The return value is the logical block number to which the pointer has been set.

CD player.seektrack(track)
Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified track. The return value is the logical block number to which the pointer has been set.

CD player.stop()
Stops the current playing operation.

CD player.togglepause()
Pauses the CD if it is playing, and makes it play if it is paused.

39.3.2 Parser Objects

Parser objects (returned by createparser()) have the following methods:

CD parser.addcallback(type, func, arg)
Adds a callback for the parser. The parser has callbacks for eight different types of data in the digital audio data stream. Constants for these types are defined at the cd module level (see above). The callback is called as follows: func(arg, type, data), where arg is the user supplied argument, type is the particular type of callback, and data is the data returned for this type of callback. The type of the data depends on the type of callback as follows:

Type	Value
audio	String which can be passed unmodified to al.writesamps().
pnum	Integer giving the program (track) number.
index	Integer giving the index number.
ptime	Tuple consisting of the program time in minutes, seconds, and frames.
atime	Tuple consisting of the absolute time in minutes, seconds, and frames.
catalog	String of 13 characters, giving the catalog number of the CD.
ident	String of 12 characters, giving the ISRC identification number of the recording. The string consists of two characters country code, three characters owner code, two characters giving the year, and five characters giving a serial number.
control	Integer giving the control bits from the CD subcode data

CD parser.deleteparser()
Deletes the parser and frees the memory it was using. The object should not be used after this call. This call is done automatically when the last reference to the object is removed.

CD parser.parseframe(frame)
Parses one or more frames of digital audio data from a CD such as returned by readda(). It determines which subcodes are present in the data. If these subcodes have changed since the last frame, then parseframe() executes a callback of the appropriate type passing to it the subcode data found in the frame. Unlike the C function, more than one frame of digital audio data can be passed to this method.

CD parser.removecallback(type)
Removes the callback for the given type.

CD parser.resetparser()
Resets the fields of the parser used for tracking subcodes to an initial state. resetparser() should be called after the disc has been changed.

39.4 fl — FORMS library for graphical user interfaces

Deprecated since version 2.6: The fl module has been removed in Python 3.

This module provides an interface to the FORMS Library by Mark Overmars. The source for the library can be retrieved by anonymous FTP from host ftp.cs.ruu.nl, directory SGI/FORMS. It was last tested with version 2.0b.

Most functions are literal translations of their C equivalents, dropping the initial fl_ from their name. Constants used by the library are defined in module FL described below.

The creation of objects is a little different in Python than in C: instead of the 'current form' maintained by the library to which new FORMS objects are added, all functions that add a FORMS object to a form are methods of the Python object representing the form. Consequently, there are no Python equivalents for the C functions fl_addto_form() and fl_end_form(), and the equivalent of fl_bgn_form() is called fl.make_form().

Watch out for the somewhat confusing terminology: FORMS uses the word object for the buttons, sliders etc. that you can place in a form. In Python, 'object' means any value. The Python interface to FORMS introduces two new Python object types: form objects (representing an entire form) and FORMS objects (representing one button, slider etc.). Hopefully this isn't too confusing.

There are no 'free objects' in the Python interface to FORMS, nor is there an easy way to add object classes written in Python. The FORMS interface to GL event handling is available, though, so you can mix FORMS with pure GL windows.

Please note: importing fl implies a call to the GL function foreground() and to the FORMS routine fl_init().

39.4.1 Functions Defined in Module fl

Module fl defines the following functions. For more information about what they do, see the description of the equivalent C function in the FORMS documentation:

fl.make_form(type, width, height)
Create a form with given type, width and height. This returns a form object, whose methods are described below.

fl.do_forms()
The standard FORMS main loop. Returns a Python object representing the FORMS object needing interaction, or the special value FL.EVENT.

fl.check_forms()
Check for FORMS events. Returns what do_forms() above returns, or None if there is no event that immediately needs interaction.

fl.set_event_call_back(function)
Set the event callback function.

fl.set_graphics_mode(rgbmode, doublebuffering)
Set the graphics modes.

fl.get_rgbmode()
: Return the current rgb mode. This is the value of the C global variable fl_rgbmode.

fl.show_message(str1, str2, str3)
: Show a dialog box with a three-line message and an OK button.

fl.show_question(str1, str2, str3)
: Show a dialog box with a three-line message and YES and NO buttons. It returns 1 if the user pressed YES, 0 if NO.

fl.show_choice(str1, str2, str3, but1[, but2[, but3]])
: Show a dialog box with a three-line message and up to three buttons. It returns the number of the button clicked by the user (1, 2 or 3).

fl.show_input(prompt, default)
: Show a dialog box with a one-line prompt message and text field in which the user can enter a string. The second argument is the default input string. It returns the string value as edited by the user.

fl.show_file_selector(message, directory, pattern, default)
: Show a dialog box in which the user can select a file. It returns the absolute filename selected by the user, or None if the user presses Cancel.

fl.get_directory()
fl.get_pattern()
fl.get_filename()
: These functions return the directory, pattern and filename (the tail part only) selected by the user in the last show_file_selector() call.

fl.qdevice(dev)
fl.unqdevice(dev)
fl.isqueued(dev)
fl.qtest()
fl.qread()
fl.qreset()
fl.qenter(dev, val)
fl.get_mouse()
fl.tie(button, valuator1, valuator2)
: These functions are the FORMS interfaces to the corresponding GL functions. Use these if you want to handle some GL events yourself when using fl.do_events(). When a GL event is detected that FORMS cannot handle, fl.do_forms() returns the special value FL.EVENT and you should call fl.qread() to read the event from the queue. Don't use the equivalent GL functions!

fl.color()
fl.mapcolor()
fl.getmcolor()
: See the description in the FORMS documentation of fl_color(), fl_mapcolor() and fl_getmcolor().

39.4.2 Form Objects

Form objects (returned by make_form() above) have the following methods. Each method corresponds to a C function whose name is prefixed with fl_; and whose first argument is a form pointer; please refer to the official FORMS documentation for descriptions.

All the add_*() methods return a Python object representing the FORMS object. Methods of FORMS objects are described below. Most kinds of FORMS object also have some methods specific to that kind; these methods are listed here.

form.show_form(placement, bordertype, name)
: Show the form.

form.hide_form()
: Hide the form.

form.redraw_form()
: Redraw the form.

form.set_form_position(x, y)
: Set the form's position.

form.freeze_form()
: Freeze the form.

form.unfreeze_form()
: Unfreeze the form.

form.activate_form()
: Activate the form.

form.deactivate_form()
: Deactivate the form.

form.bgn_group()
: Begin a new group of objects; return a group object.

form.end_group()
: End the current group of objects.

form.find_first()
: Find the first object in the form.

form.find_last()
: Find the last object in the form.

form.add_box(type, x, y, w, h, name)
: Add a box object to the form. No extra methods.

form.add_text(type, x, y, w, h, name)
: Add a text object to the form. No extra methods.

form.add_clock(type, x, y, w, h, name)
: Add a clock object to the form. — Method: get_clock().

form.add_button(type, x, y, w, h, name)
: Add a button object to the form. — Methods: get_button(), set_button().

form.add_lightbutton(type, x, y, w, h, name)
: Add a lightbutton object to the form. — Methods: get_button(), set_button().

form.add_roundbutton(type, x, y, w, h, name)
: Add a roundbutton object to the form. — Methods: get_button(), set_button().

form.add_slider(type, x, y, w, h, name)
: Add a slider object to the form. — Methods: set_slider_value(), get_slider_value(), set_slider_bounds(), get_slider_bounds(), set_slider_return(), set_slider_size(), set_slider_precision(), set_slider_step().

form.add_valslider(type, x, y, w, h, name)
: Add a valslider object to the form. — Methods: set_slider_value(), get_slider_value(), set_slider_bounds(), get_slider_bounds(), set_slider_return(), set_slider_size(), set_slider_precision(), set_slider_step().

form.add_dial(type, x, y, w, h, name)
Add a dial object to the form. — Methods: set_dial_value(), get_dial_value(), set_dial_bounds(), get_dial_bounds().

form.add_positioner(type, x, y, w, h, name)
Add a positioner object to the form. — Methods: set_positioner_xvalue(), set_positioner_yvalue(), set_positioner_xbounds(), set_positioner_ybounds(), get_positioner_xvalue(), get_positioner_yvalue(), get_positioner_xbounds(), get_positioner_ybounds().

form.add_counter(type, x, y, w, h, name)
Add a counter object to the form. — Methods: set_counter_value(), get_counter_value(), set_counter_bounds(), set_counter_step(), set_counter_precision(), set_counter_return().

form.add_input(type, x, y, w, h, name)
Add an input object to the form. — Methods: set_input(), get_input(), set_input_color(), set_input_return().

form.add_menu(type, x, y, w, h, name)
Add a menu object to the form. — Methods: set_menu(), get_menu(), addto_menu().

form.add_choice(type, x, y, w, h, name)
Add a choice object to the form. — Methods: set_choice(), get_choice(), clear_choice(), addto_choice(), replace_choice(), delete_choice(), get_choice_text(), set_choice_fontsize(), set_choice_fontstyle().

form.add_browser(type, x, y, w, h, name)
Add a browser object to the form. — Methods: set_browser_topline(), clear_browser(), add_browser_line(), addto_browser(), insert_browser_line(), delete_browser_line(), replace_browser_line(), get_browser_line(), load_browser(), get_browser_maxline(), select_browser_line(), deselect_browser_line(), deselect_browser(), isselected_browser_line(), get_browser(), set_browser_fontsize(), set_browser_fontstyle(), set_browser_specialkey().

form.add_timer(type, x, y, w, h, name)
Add a timer object to the form. — Methods: set_timer(), get_timer().

Form objects have the following data attributes; see the FORMS documentation:

Name	C Type	Meaning
window	int (read-only)	GL window id
w	float	form width
h	float	form height
x	float	form x origin
y	float	form y origin
deactivated	int	nonzero if form is deactivated
visible	int	nonzero if form is visible
frozen	int	nonzero if form is frozen
doublebuf	int	nonzero if double buffering on

39.4.3 FORMS Objects

Besides methods specific to particular kinds of FORMS objects, all FORMS objects also have the following methods:

FORMS object.set_call_back(function, argument)
Set the object's callback function and argument. When the object needs interaction, the callback

function will be called with two arguments: the object, and the callback argument. (FORMS objects without a callback function are returned by fl.do_forms() or fl.check_forms() when they need interaction.) Call this method without arguments to remove the callback function.

FORMS object.delete_object()
Delete the object.

FORMS object.show_object()
Show the object.

FORMS object.hide_object()
Hide the object.

FORMS object.redraw_object()
Redraw the object.

FORMS object.freeze_object()
Freeze the object.

FORMS object.unfreeze_object()
Unfreeze the object.

FORMS objects have these data attributes; see the FORMS documentation:

Name	C Type	Meaning
objclass	int (read-only)	object class
type	int (read-only)	object type
boxtype	int	box type
x	float	x origin
y	float	y origin
w	float	width
h	float	height
col1	int	primary color
col2	int	secondary color
align	int	alignment
lcol	int	label color
lsize	float	label font size
label	string	label string
lstyle	int	label style
pushed	int (read-only)	(see FORMS docs)
focus	int (read-only)	(see FORMS docs)
belowmouse	int (read-only)	(see FORMS docs)
frozen	int (read-only)	(see FORMS docs)
active	int (read-only)	(see FORMS docs)
input	int (read-only)	(see FORMS docs)
visible	int (read-only)	(see FORMS docs)
radio	int (read-only)	(see FORMS docs)
automatic	int (read-only)	(see FORMS docs)

39.5 FL — Constants used with the fl module

Deprecated since version 2.6: The FL module has been removed in Python 3.

This module defines symbolic constants needed to use the built-in module fl (see above); they are equivalent to those defined in the C header file <forms.h> except that the name prefix FL_ is omitted. Read the

module source for a complete list of the defined names. Suggested use:

```
import fl
from FL import *
```

39.6 flp — Functions for loading stored FORMS designs

Deprecated since version 2.6: The flp module has been removed in Python 3.

This module defines functions that can read form definitions created by the 'form designer' (fdesign) program that comes with the FORMS library (see module fl above).

For now, see the file flp.doc in the Python library source directory for a description.

XXX A complete description should be inserted here!

39.7 fm — Font Manager interface

Deprecated since version 2.6: The fm module has been removed in Python 3.

This module provides access to the IRIS Font Manager library. It is available only on Silicon Graphics machines. See also: 4Sight User's Guide, section 1, chapter 5: "Using the IRIS Font Manager."

This is not yet a full interface to the IRIS Font Manager. Among the unsupported features are: matrix operations; cache operations; character operations (use string operations instead); some details of font info; individual glyph metrics; and printer matching.

It supports the following operations:

fm.init()
: Initialization function. Calls fminit(). It is normally not necessary to call this function, since it is called automatically the first time the fm module is imported.

fm.findfont(fontname)
: Return a font handle object. Calls fmfindfont(fontname).

fm.enumerate()
: Returns a list of available font names. This is an interface to fmenumerate().

fm.prstr(string)
: Render a string using the current font (see the setfont() font handle method below). Calls fmprstr(string).

fm.setpath(string)
: Sets the font search path. Calls fmsetpath(string). (XXX Does not work!?!)

fm.fontpath()
: Returns the current font search path.

Font handle objects support the following operations:

font handle.scalefont(factor)
: Returns a handle for a scaled version of this font. Calls fmscalefont(fh, factor).

font handle.setfont()
: Makes this font the current font. Note: the effect is undone silently when the font handle object is deleted. Calls fmsetfont(fh).

font handle.getfontname()
Returns this font's name. Calls fmgetfontname(fh).

font handle.getcomment()
Returns the comment string associated with this font. Raises an exception if there is none. Calls fmgetcomment(fh).

font handle.getfontinfo()
Returns a tuple giving some pertinent data about this font. This is an interface to fmgetfontinfo(). The returned tuple contains the following numbers: (printermatched, fixed_width, xorig, yorig, xsize, ysize, height, nglyphs).

font handle.getstrwidth(string)
Returns the width, in pixels, of string when drawn in this font. Calls fmgetstrwidth(fh, string).

39.8 gl — Graphics Library interface

Deprecated since version 2.6: The gl module has been removed in Python 3.

This module provides access to the Silicon Graphics Graphics Library. It is available only on Silicon Graphics machines.

Warning: Some illegal calls to the GL library cause the Python interpreter to dump core. In particular, the use of most GL calls is unsafe before the first window is opened.

The module is too large to document here in its entirety, but the following should help you to get started. The parameter conventions for the C functions are translated to Python as follows:

- All (short, long, unsigned) int values are represented by Python integers.
- All float and double values are represented by Python floating point numbers. In most cases, Python integers are also allowed.
- All arrays are represented by one-dimensional Python lists. In most cases, tuples are also allowed.
- All string and character arguments are represented by Python strings, for instance, winopen('Hi There!') and rotate(900, 'z').
- All (short, long, unsigned) integer arguments or return values that are only used to specify the length of an array argument are omitted. For example, the C call

```
lmdef(deftype, index, np, props)
```

  is translated to Python as

```
lmdef(deftype, index, props)
```

- Output arguments are omitted from the argument list; they are transmitted as function return values instead. If more than one value must be returned, the return value is a tuple. If the C function has both a regular return value (that is not omitted because of the previous rule) and an output argument, the return value comes first in the tuple. Examples: the C call

```
getmcolor(i, &red, &green, &blue)
```

  is translated to Python as

```
red, green, blue = getmcolor(i)
```

The following functions are non-standard or have special argument conventions:

gl.varray(argument)
: Equivalent to but faster than a number of v3d() calls. The argument is a list (or tuple) of points. Each point must be a tuple of coordinates (x, y, z) or (x, y). The points may be 2- or 3-dimensional but must all have the same dimension. Float and int values may be mixed however. The points are always converted to 3D double precision points by assuming z = 0.0 if necessary (as indicated in the man page), and for each point v3d() is called.

gl.nvarray()
: Equivalent to but faster than a number of n3f and v3f calls. The argument is an array (list or tuple) of pairs of normals and points. Each pair is a tuple of a point and a normal for that point. Each point or normal must be a tuple of coordinates (x, y, z). Three coordinates must be given. Float and int values may be mixed. For each pair, n3f() is called for the normal, and then v3f() is called for the point.

gl.vnarray()
: Similar to nvarray() but the pairs have the point first and the normal second.

gl.nurbssurface(s_k, t_k, ctl, s_ord, t_ord, type)
: Defines a nurbs surface. The dimensions of ctl[][][] are computed as follows: [len(s_k) - s_ord], [len(t_k) - t_ord].

gl.nurbscurve(knots, ctlpoints, order, type)
: Defines a nurbs curve. The length of ctlpoints is len(knots) - order.

gl.pwlcurve(points, type)
: Defines a piecewise-linear curve. points is a list of points. type must be N_ST.

gl.pick(n)
gl.select(n)
: The only argument to these functions specifies the desired size of the pick or select buffer.

gl.endpick()
gl.endselect()
: These functions have no arguments. They return a list of integers representing the used part of the pick/select buffer. No method is provided to detect buffer overrun.

Here is a tiny but complete example GL program in Python:

```
import gl, GL, time

def main():
    gl.foreground()
    gl.prefposition(500, 900, 500, 900)
    w = gl.winopen('CrissCross')
    gl.ortho2(0.0, 400.0, 0.0, 400.0)
    gl.color(GL.WHITE)
    gl.clear()
    gl.color(GL.RED)
    gl.bgnline()
    gl.v2f(0.0, 0.0)
    gl.v2f(400.0, 400.0)
    gl.endline()
    gl.bgnline()
    gl.v2f(400.0, 0.0)
    gl.v2f(0.0, 400.0)
    gl.endline()
```

(continues on next page)

(continued from previous page)

```
    time.sleep(5)

main()
```

See also:

PyOpenGL: The Python OpenGL Binding An interface to OpenGL is also available; see information about the PyOpenGL project online at http://pyopengl.sourceforge.net/. This may be a better option if support for SGI hardware from before about 1996 is not required.

39.9 DEVICE — Constants used with the gl module

Deprecated since version 2.6: The DEVICE module has been removed in Python 3.

This modules defines the constants used by the Silicon Graphics Graphics Library that C programmers find in the header file <gl/device.h>. Read the module source file for details.

39.10 GL — Constants used with the gl module

Deprecated since version 2.6: The GL module has been removed in Python 3.

This module contains constants used by the Silicon Graphics Graphics Library from the C header file <gl/gl.h>. Read the module source file for details.

39.11 imgfile — Support for SGI imglib files

Deprecated since version 2.6: The imgfile module has been removed in Python 3.

The imgfile module allows Python programs to access SGI imglib image files (also known as .rgb files). The module is far from complete, but is provided anyway since the functionality that there is enough in some cases. Currently, colormap files are not supported.

The module defines the following variables and functions:

exception imgfile.error
: This exception is raised on all errors, such as unsupported file type, etc.

imgfile.getsizes(file)
: This function returns a tuple (x, y, z) where x and y are the size of the image in pixels and z is the number of bytes per pixel. Only 3 byte RGB pixels and 1 byte greyscale pixels are currently supported.

imgfile.read(file)
: This function reads and decodes the image on the specified file, and returns it as a Python string. The string has either 1 byte greyscale pixels or 4 byte RGBA pixels. The bottom left pixel is the first in the string. This format is suitable to pass to gl.lrectwrite(), for instance.

imgfile.readscaled(file, x, y, filter[, blur])
: This function is identical to read but it returns an image that is scaled to the given x and y sizes. If the filter and blur parameters are omitted scaling is done by simply dropping or duplicating pixels, so the result will be less than perfect, especially for computer-generated images.

Alternatively, you can specify a filter to use to smooth the image after scaling. The filter forms supported are `'impulse'`, `'box'`, `'triangle'`, `'quadratic'` and `'gaussian'`. If a filter is specified blur is an optional parameter specifying the blurriness of the filter. It defaults to 1.0.

readscaled() makes no attempt to keep the aspect ratio correct, so that is the users' responsibility.

imgfile.ttob(flag)
: This function sets a global flag which defines whether the scan lines of the image are read or written from bottom to top (flag is zero, compatible with SGI GL) or from top to bottom(flag is one, compatible with X). The default is zero.

imgfile.write(file, data, x, y, z)
: This function writes the RGB or greyscale data in data to image file file. x and y give the size of the image, z is 1 for 1 byte greyscale images or 3 for RGB images (which are stored as 4 byte values of which only the lower three bytes are used). These are the formats returned by gl.lrectread().

39.12 jpeg — Read and write JPEG files

Deprecated since version 2.6: The jpeg module has been removed in Python 3.

The module jpeg provides access to the jpeg compressor and decompressor written by the Independent JPEG Group (IJG). JPEG is a standard for compressing pictures; it is defined in ISO 10918. For details on JPEG or the Independent JPEG Group software refer to the JPEG standard or the documentation provided with the software.

A portable interface to JPEG image files is available with the Python Imaging Library (PIL) by Fredrik Lundh. Information on PIL is available at http://www.pythonware.com/products/pil/.

The jpeg module defines an exception and some functions.

exception jpeg.error
: Exception raised by compress() and decompress() in case of errors.

jpeg.compress(data, w, h, b)
: Treat data as a pixmap of width w and height h, with b bytes per pixel. The data is in SGI GL order, so the first pixel is in the lower-left corner. This means that gl.lrectread() return data can immediately be passed to compress(). Currently only 1 byte and 4 byte pixels are allowed, the former being treated as greyscale and the latter as RGB color. compress() returns a string that contains the compressed picture, in JFIF format.

jpeg.decompress(data)
: Data is a string containing a picture in JFIF format. It returns a tuple (data, width, height, bytesperpixel). Again, the data is suitable to pass to gl.lrectwrite().

jpeg.setoption(name, value)
: Set various options. Subsequent compress() and decompress() calls will use these options. The following options are available:

Option	Effect
`'forcegray'`	Force output to be grayscale, even if input is RGB.
`'quality'`	Set the quality of the compressed image to a value between 0 and 100 (default is 75). This only affects compression.
`'optimize'`	Perform Huffman table optimization. Takes longer, but results in smaller compressed image. This only affects compression.
`'smooth'`	Perform inter-block smoothing on uncompressed image. Only useful for low- quality images. This only affects decompression.

See also:

JPEG Still Image Data Compression Standard The canonical reference for the JPEG image format, by Pennebaker and Mitchell.

Information Technology - Digital Compression and Coding of Continuous-tone Still Images - Requirements and Guidelines The ISO standard for JPEG is also published as ITU T.81. This is available online in PDF form.

CHAPTER

FORTY

SUNOS SPECIFIC SERVICES

The modules described in this chapter provide interfaces to features that are unique to SunOS 5 (also known as Solaris version 2).

40.1 sunaudiodev — Access to Sun audio hardware

Deprecated since version 2.6: The sunaudiodev module has been removed in Python 3.

This module allows you to access the Sun audio interface. The Sun audio hardware is capable of recording and playing back audio data in u-LAW format with a sample rate of 8K per second. A full description can be found in the audio(7I) manual page.

The module SUNAUDIODEV defines constants which may be used with this module.

This module defines the following variables and functions:

exception sunaudiodev.error
: This exception is raised on all errors. The argument is a string describing what went wrong.

sunaudiodev.open(mode)
: This function opens the audio device and returns a Sun audio device object. This object can then be used to do I/O on. The mode parameter is one of `'r'` for record-only access, `'w'` for play-only access, `'rw'` for both and `'control'` for access to the control device. Since only one process is allowed to have the recorder or player open at the same time it is a good idea to open the device only for the activity needed. See audio(7I) for details.

  As per the manpage, this module first looks in the environment variable AUDIODEV for the base audio device filename. If not found, it falls back to /dev/audio. The control device is calculated by appending "ctl" to the base audio device.

40.1.1 Audio Device Objects

The audio device objects are returned by open() define the following methods (except control objects which only provide getinfo(), setinfo(), fileno(), and drain()):

audio device.close()
: This method explicitly closes the device. It is useful in situations where deleting the object does not immediately close it since there are other references to it. A closed device should not be used again.

audio device.fileno()
: Returns the file descriptor associated with the device. This can be used to set up SIGPOLL notification, as described below.

audio device.drain()
: This method waits until all pending output is processed and then returns. Calling this method is often not necessary: destroying the object will automatically close the audio device and this will do an implicit drain.

audio device.flush()
: This method discards all pending output. It can be used avoid the slow response to a user's stop request (due to buffering of up to one second of sound).

audio device.getinfo()
: This method retrieves status information like input and output volume, etc. and returns it in the form of an audio status object. This object has no methods but it contains a number of attributes describing the current device status. The names and meanings of the attributes are described in <sun/audioio.h> and in the audio(7I) manual page. Member names are slightly different from their C counterparts: a status object is only a single structure. Members of the play substructure have o_ prepended to their name and members of the record structure have i_. So, the C member play.sample_rate is accessed as o_sample_rate, record.gain as i_gain and monitor_gain plainly as monitor_gain.

audio device.ibufcount()
: This method returns the number of samples that are buffered on the recording side, i.e. the program will not block on a read() call of so many samples.

audio device.obufcount()
: This method returns the number of samples buffered on the playback side. Unfortunately, this number cannot be used to determine a number of samples that can be written without blocking since the kernel output queue length seems to be variable.

audio device.read(size)
: This method reads size samples from the audio input and returns them as a Python string. The function blocks until enough data is available.

audio device.setinfo(status)
: This method sets the audio device status parameters. The status parameter is a device status object as returned by getinfo() and possibly modified by the program.

audio device.write(samples)
: Write is passed a Python string containing audio samples to be played. If there is enough buffer space free it will immediately return, otherwise it will block.

The audio device supports asynchronous notification of various events, through the SIGPOLL signal. Here's an example of how you might enable this in Python:

```
def handle_sigpoll(signum, frame):
    print 'I got a SIGPOLL update'

import fcntl, signal, STROPTS

signal.signal(signal.SIGPOLL, handle_sigpoll)
fcntl.ioctl(audio_obj.fileno(), STROPTS.I_SETSIG, STROPTS.S_MSG)
```

40.2 SUNAUDIODEV — Constants used with sunaudiodev

Deprecated since version 2.6: The SUNAUDIODEV module has been removed in Python 3.

This is a companion module to sunaudiodev which defines useful symbolic constants like MIN_GAIN, MAX_GAIN, SPEAKER, etc. The names of the constants are the same names as used in the C include file <sun/audioio.h>, with the leading string AUDIO_ stripped.

CHAPTER

FORTYONE

UNDOCUMENTED MODULES

Here's a quick listing of modules that are currently undocumented, but that should be documented. Feel free to contribute documentation for them! (Send via email to docs@python.org.)

The idea and original contents for this chapter were taken from a posting by Fredrik Lundh; the specific contents of this chapter have been substantially revised.

41.1 Miscellaneous useful utilities

Some of these are very old and/or not very robust; marked with "hmm."

ihooks — Import hook support (for rexec; may become obsolete). Removed in Python 3.x.

41.2 Platform specific modules

These modules are used to implement the os.path module, and are not documented beyond this mention. There's little need to document these.

ntpath — Implementation of os.path on Win32, Win64, WinCE, and OS/2 platforms.

posixpath — Implementation of os.path on POSIX.

bsddb185 — Backwards compatibility module for systems which still use the Berkeley DB 1.85 module. It is normally only available on certain BSD Unix-based systems. It should never be used directly.

41.3 Multimedia

audiodev — Platform-independent API for playing audio data. Removed in Python 3.x.

linuxaudiodev — Play audio data on the Linux audio device. Replaced in Python 2.3 by the ossaudiodev module. Removed in Python 3.x.

sunaudio — Interpret Sun audio headers (may become obsolete or a tool/demo). Removed in Python 3.x.

toaiff — Convert "arbitrary" sound files to AIFF files; should probably become a tool or demo. Requires the external program sox. Removed in Python 3.x.

41.4 Undocumented Mac OS modules

41.4.1 applesingle — AppleSingle decoder

Deprecated since version 2.6.

41.4.2 buildtools — Helper module for BuildApplet and Friends

Deprecated since version 2.4.

41.4.3 cfmfile — Code Fragment Resource module

cfmfile is a module that understands Code Fragments and the accompanying "cfrg" resources. It can parse them and merge them, and is used by BuildApplication to combine all plugin modules to a single executable.

Deprecated since version 2.4.

41.4.4 icopen — Internet Config replacement for open()

Importing icopen will replace the built-in open() with a version that uses Internet Config to set file type and creator for new files.

Deprecated since version 2.6.

41.4.5 macerrors — Mac OS Errors

macerrors contains constant definitions for many Mac OS error codes.

Deprecated since version 2.6.

41.4.6 macresource — Locate script resources

macresource helps scripts finding their resources, such as dialogs and menus, without requiring special case code for when the script is run under MacPython, as a MacPython applet or under OSX Python.

Deprecated since version 2.6.

41.4.7 Nav — NavServices calls

A low-level interface to Navigation Services.

Deprecated since version 2.6.

41.4.8 PixMapWrapper — Wrapper for PixMap objects

PixMapWrapper wraps a PixMap object with a Python object that allows access to the fields by name. It also has methods to convert to and from PIL images.

Deprecated since version 2.6.

41.4.9 videoreader — Read QuickTime movies

videoreader reads and decodes QuickTime movies and passes a stream of images to your program. It also provides some support for audio tracks.

Deprecated since version 2.6.

41.4.10 W — Widgets built on FrameWork

The W widgets are used extensively in the IDE.

Deprecated since version 2.6.

41.5 Obsolete

These modules are not normally available for import; additional work must be done to make them available.

These extension modules written in C are not built by default. Under Unix, these must be enabled by uncommenting the appropriate lines in Modules/Setup in the build tree and either rebuilding Python if the modules are statically linked, or building and installing the shared object if using dynamically-loaded extensions.

timing — Measure time intervals to high resolution (use time.clock() instead). Removed in Python 3.x.

41.6 SGI-specific Extension modules

The following are SGI specific, and may be out of touch with the current version of reality.

cl — Interface to the SGI compression library.

sv — Interface to the "simple video" board on SGI Indigo (obsolete hardware). Removed in Python 3.x.

APPENDIX

A

GLOSSARY

>>> The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter.

... The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

2to3 A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as lib2to3; a standalone entry point is provided as Tools/scripts/2to3. See 2to3 - Automated Python 2 to 3 code translation.

abstract base class Abstract base classes complement duck-typing by providing a way to define interfaces when other techniques like hasattr() would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by isinstance() and issubclass(); see the abc module documentation. Python comes with many built-in ABCs for data structures (in the collections module), numbers (in the numbers module), and streams (in the io module). You can create your own ABCs with the abc module.

argument A value passed to a function (or method) when calling the function. There are two types of arguments:

- keyword argument: an argument preceded by an identifier (e.g. name=) in a function call or passed as a value in a dictionary preceded by **. For example, 3 and 5 are both keyword arguments in the following calls to complex():

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- positional argument: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an iterable preceded by *. For example, 3 and 5 are both positional arguments in the following calls:

```
complex(3, 5)
complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body. See the calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the parameter glossary entry and the FAQ question on the difference between arguments and parameters.

attribute A value associated with an object which is referenced by name using dotted expressions. For example, if an object o has an attribute a it would be referenced as o.a.

BDFL Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

bytes-like object An object that supports the buffer protocol, like str, bytearray or memoryview. Bytes-like objects can be used for various operations that expect binary data, such as compression, saving to a binary file or sending over a socket. Some operations need the binary data to be mutable, in which case not all bytes-like objects can apply.

bytecode Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in .pyc and .pyo files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This "intermediate language" is said to run on a virtual machine that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the dis module.

class A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

classic class Any class which does not inherit from object. See new-style class. Classic classes have been removed in Python 3.

coercion The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, int(3.15) converts the floating point number to the integer 3, but in 3+4.5, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a TypeError. Coercion between two operands can be performed with the coerce built-in function; thus, 3+4.5 is equivalent to calling operator.add(*coerce(3, 4.5)) and results in operator.add(3.0, 4.5). Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., float(3)+4.5 rather than just 3+4.5.

complex number An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of -1), often written i in mathematics or j in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a j suffix, e.g., 3+1j. To get access to complex equivalents of the math module, use cmath. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

context manager An object which controls the environment seen in a with statement by defining __enter__() and __exit__() methods. See PEP 343.

CPython The canonical implementation of the Python programming language, as distributed on python.org. The term "CPython" is used when necessary to distinguish this implementation from others such as Jython or IronPython.

decorator A function returning another function, usually applied as a function transformation using the @wrapper syntax. Common examples for decorators are classmethod() and staticmethod().

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for function definitions and class definitions for more about decorators.

descriptor Any new-style object which defines the methods __get__(), __set__(), or __delete__(). When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using a.b to get, set or delete an attribute looks up the object named b in the class dictionary for a, but if b is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors' methods, see descriptors.

dictionary An associative array, where arbitrary keys are mapped to values. The keys can be any object with __hash__() and __eq__() methods. Called a hash in Perl.

dictionary view The objects returned from dict.viewkeys(), dict.viewvalues(), and dict.viewitems() are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use list(dictview). See Dictionary view objects.

docstring A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the __doc__ attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using type() or isinstance(). (Note, however, that duck-typing can be complemented with abstract base classes.) Instead, it typically employs hasattr() tests or EAFP programming.

EAFP Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements. The technique contrasts with the LBYL style common to many other languages such as C.

expression A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also statements which cannot be used as expressions, such as print or if. Assignments are also statements, not expressions.

extension module A module written in C or C++, using Python's C API to interact with the core and with user code.

file object An object exposing a file-oriented API (with methods such as read() or write()) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called file-like objects or streams.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the io module. The canonical way to create a file object is by using the open() function.

file-like object A synonym for file object.

finder An object that tries to find the loader for a module. It must implement a method named find_module(). See PEP 302 for details.

floor division Mathematical division that rounds down to nearest integer. The floor division operator is //. For example, the expression 11 // 4 evaluates to 2 in contrast to the 2.75 returned by float true division. Note that (-11) // 4 is -3 because that is -2.75 rounded downward. See PEP 238.

function A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body. See also parameter, method, and the function section.

__future__ A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression 11/4 currently evaluates to 2. If the module in which it is executed had enabled true division by executing:

```
from __future__ import division
```

the expression 11/4 would evaluate to 2.75. By importing the __future__ module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

generator A function which returns an iterator. It looks like a normal function except that it contains yield statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the next() function. Each yield temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression An expression that returns an iterator. It looks like a normal expression followed by a for expression defining a loop variable, range, and an optional if expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

GIL See global interpreter lock.

global interpreter lock The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as dict) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

Past efforts to create a "free-threaded" interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.

hashable An object is hashable if it has a hash value which never changes during its lifetime (it needs a __hash__() method), and can be compared to other objects (it needs an __eq__() or __cmp__() method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is derived from their id().

IDLE An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.

immutable An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

integer division Mathematical division discarding any remainder. For example, the expression 11/4 currently evaluates to 2 in contrast to the 2.75 returned by float division. Also called floor division. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a float), the result will be coerced (see coercion) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the // operator instead of the / operator. See also __future__.

importing The process by which Python code in one module is made available to Python code in another module.

importer An object that both finds and loads a module; both a finder and loader object.

interactive Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch python with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember help(x)).

interpreted Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also interactive.

iterable An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as list, str, and tuple) and some non-sequence types like dict and file and objects of any classes you define with an __iter__() or __getitem__() method. Iterables can be used in a for loop and in many other places where a sequence is needed (zip(), map(), ...). When an iterable object is passed as an argument to the built-in function iter(), it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call iter() or deal with iterator objects yourself. The for statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also iterator, sequence, and generator.

iterator An object representing a stream of data. Repeated calls to the iterator's next() method return successive items in the stream. When no more data are available a StopIteration exception is raised instead. At this point, the iterator object is exhausted and any further calls to its next() method just raise StopIteration again. Iterators are required to have an __iter__() method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a list) produces a fresh new iterator each time you pass it to the iter() function or use it in a for loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in Iterator Types.

key function A key function or collation function is a callable that returns a value used for sorting or ordering. For example, locale.strxfrm() is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include min(), max(), sorted(), list.sort(), heapq.nsmallest(), heapq.nlargest(), and itertools.groupby().

There are several ways to create a key function. For example. the str.lower() method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a lambda expression such as lambda r: (r[0], r[2]). Also, the operator module provides three key function constructors: attrgetter(), itemgetter(), and methodcaller(). See the Sorting HOW TO for examples of how to create and use key functions.

keyword argument See argument.

lambda An anonymous inline function consisting of a single expression which is evaluated when the function is called. The syntax to create a lambda function is lambda [parameters]: expression

LBYL Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many if statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between "the looking" and "the leaping". For example, the code, if key in mapping: return mapping[key] can fail if another thread removes key from mapping after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

list A built-in Python sequence. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is O(1).

list comprehension A compact way to process all or part of the elements in a sequence and return a list with the results. result = ["0x%02x" % x for x in range(256) if x % 2 == 0] generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The if clause is optional. If omitted, all elements in range(256) are processed.

loader An object that loads a module. It must define a method named load_module(). A loader is typically returned by a finder. See PEP 302 for details.

mapping A container object that supports arbitrary key lookups and implements the methods specified in the Mapping or MutableMapping abstract base classes. Examples include dict, collections.defaultdict, collections.OrderedDict and collections.Counter.

metaclass The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in metaclasses.

method A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called self). See function and nested scope.

method resolution order Method Resolution Order is the order in which base classes are searched for a member during lookup. See The Python 2.3 Method Resolution Order for details of the algorithm used by the Python interpreter since the 2.3 release.

module An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of importing.

See also package.

MRO See method resolution order.

mutable Mutable objects can change their value but keep their id(). See also immutable.

named tuple Any tuple-like class whose indexable elements are also accessible using named attributes (for example, time.localtime() returns a tuple-like object where the year is accessible either with an index such as t[0] or with a named attribute like t.tm_year).

A named tuple can be a built-in type such as time.struct_time, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function collections.namedtuple(). The latter approach automatically provides extra features such as a self-documenting representation like Employee(name='jones', title='programmer').

namespace The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions __builtin__.open() and os.open() are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing random.seed() or itertools.izip() makes it clear that those functions are implemented by the random and itertools modules, respectively.

nested scope The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

new-style class Any class which inherits from object. This includes all built-in types like list and dict. Only new-style classes can use Python's newer, versatile features like __slots__, descriptors, properties, and __getattribute__().

More information can be found in newstyle.

object Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any new-style class.

package A Python module which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an __path__ attribute.

parameter A named entity in a function (or method) definition that specifies an argument (or in some cases, arguments) that the function can accept. There are four types of parameters:

- positional-or-keyword: specifies an argument that can be passed either positionally or as a keyword argument. This is the default kind of parameter, for example foo and bar in the following:

```
def func(foo, bar=None): ...
```

- positional-only: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. abs()).
- var-positional: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with *, for example args in the following:

```
def func(*args, **kwargs): ...
```

- var-keyword: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with **, for example kwargs in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the argument glossary entry, the FAQ question on the difference between arguments and parameters, and the function section.

PEP Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See PEP 1.

positional argument See argument.

Python 3000 Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated "Py3k".

Pythonic An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a for statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):
    print food[i]
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
    print piece
```

reference count The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Reference counting is generally not visible to Python code, but it is a key element of the CPython implementation. The sys module defines a getrefcount() function that programmers can call to return the reference count for a particular object.

__slots__ A declaration inside a new-style class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence An iterable which supports efficient element access using integer indices via the __getitem__() special method and defines a len() method that returns the length of the sequence. Some built-in sequence types are list, str, tuple, and unicode. Note that dict also supports __getitem__() and __len__(), but is considered a mapping rather than a sequence because the lookups use arbitrary immutable keys rather than integers.

slice An object usually containing a portion of a sequence. A slice is created using the subscript notation, [] with colons between numbers when several are given, such as in variable_name[1:3:5]. The

bracket (subscript) notation uses slice objects internally (or in older versions, __getslice__() and __setslice__()).

special method A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in specialnames.

statement A statement is part of a suite (a "block" of code). A statement is either an expression or one of several constructs with a keyword, such as if, while or for.

struct sequence A tuple with named elements. Struct sequences expose an interface similiar to named tuple in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like _make() or _asdict(). Examples of struct sequences include sys.float_info and the return value of os.stat().

triple-quoted string A string which is bound by three instances of either a quotation mark (") or an apostrophe ('). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its __class__ attribute or can be retrieved with type(obj).

universal newlines A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention '\n', the Windows convention '\r\n', and the old Macintosh convention '\r'. See PEP 278 and PEP 3116, as well as str.splitlines() for an additional use.

virtual environment A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

virtual machine A computer defined entirely in software. Python's virtual machine executes the bytecode emitted by the bytecode compiler.

Zen of Python Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing "import this" at the interactive prompt.

APPENDIX

B

ABOUT THESE DOCUMENTS

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

Development of the documentation and its toolchain is an entirely volunteer effort, just like Python itself. If you want to contribute, please take a look at the reporting-bugs page for information on how to do so. New volunteers are always welcome!

Many thanks go to:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content;
- the Docutils project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

B.1 Contributors to the Python Documentation

Many people have contributed to the Python language, the Python standard library, and the Python documentation. See Misc/ACKS in the Python source distribution for a partial list of contributors.

It is only with the input and contributions of the Python community that Python has such wonderful documentation – Thank You!

APPENDIX

C

HISTORY AND LICENSE

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 and above	2.1.1	2001-now	PSF	yes

Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 2.7.16

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.7.16 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.7.16 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2019 Python Software Foundation; All Rights Reserved" are retained in Python 2.7.16 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.7.16 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.7.16.

4. PSF is making Python 2.7.16 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY␣
   ↪REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR␣
   ↪THAT THE
   USE OF PYTHON 2.7.16 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.16
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
   ↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.16, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.7.16, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION
   OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee. This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party. As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version

(continues on next page)

(continued from previous page)

prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright

(continues on next page)

(continued from previous page)

```
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The _random module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  1. Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

  2. Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

  3. The names of its contributors may not be used to endorse or promote
     products derived from this software without specific prior written
     permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
```

(continues on next page)

(continued from previous page)

```
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

The socket module uses the functions, getaddrinfo(), and getnameinfo(), which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Floating point exception control

The source for the fpectl module includes the following notice:

```
  ---------------------------------------------------------------------
 /                     Copyright (c) 1996.                           \
|          The Regents of the University of California.               |
|                     All rights reserved.                            |
|                                                                     |
```

(continues on next page)

(continued from previous page)

```
|  Permission to use, copy, modify, and distribute this software for  |
|  any purpose without fee is hereby granted, provided that this en-  |
|  tire notice is included in all copies of any software which is or  |
|  includes a copy or modification of this software and in all  |
|  copies of the supporting documentation for such software.  |
|                                                                     |
|  This work was produced at the University of California, Lawrence  |
|  Livermore National Laboratory under contract no. W-7405-ENG-48  |
|  between the U.S. Department of Energy and The Regents of the  |
|  University of California for the operation of UC LLNL.  |
|                                                                     |
|                              DISCLAIMER                             |
|                                                                     |
|  This software was prepared as an account of work sponsored by an  |
|  agency of the United States Government. Neither the United States  |
|  Government nor the University of California nor any of their em-  |
|  ployees, makes any warranty, express or implied, or assumes any  |
|  liability or responsibility for the accuracy, completeness, or  |
|  usefulness of any information, apparatus, product, or process  |
|  disclosed, or represents that its use would not infringe  |
|  privately-owned rights. Reference herein to any specific commer-  |
|  cial products, process, or service by trade name, trademark,  |
|  manufacturer, or otherwise, does not necessarily constitute or  |
|  imply its endorsement, recommendation, or favoring by the United  |
|  States Government or the University of California. The views and  |
|  opinions of authors expressed herein do not necessarily state or  |
|  reflect those of the United States Government or the University  |
|  of California, and shall not be used for advertising or product  |
\  endorsement purposes.                                             /
 ---------------------------------------------------------------------
```

C.3.4 MD5 message digest algorithm

The source code for the md5 module contains the following notice:

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com
```

(continues on next page)

(continued from previous page)

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
    http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A. It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>. Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
    references to Ghostscript; clarified derivation from RFC 1321;
    now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
    added conditionalization for C++ compilation from Martin
    Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.

C.3.5 Asynchronous socket services

The asynchat and asyncore modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.6 Cookie management

The Cookie module contains the following notice:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.

C.3.7 Execution tracing

The trace module contains the following notice:

portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.

C.3.8 UUencode and UUdecode functions

The uu module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                        All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.9 XML Remote Procedure Calls

The xmlrpclib module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(continues on next page)

(continued from previous page)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.10 test_epoll

The test_epoll contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.11 Select kqueue

The select and contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

(continues on next page)

(continued from previous page)

```
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.12 strtod and dtoa

The file Python/dtoa.c, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from http://www.netlib.org/fp/. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.13 OpenSSL

The modules hashlib, posix, ssl, crypt use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /*
 ↪====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
```

(continues on next page)

(continued from previous page)

```
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
   * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
   * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
   * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
   * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
   * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
   * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
   * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
   * OF THE POSSIBILITY OF SUCH DAMAGE.
   *␣
↪==========================================================================
   *
   * This product includes cryptographic software written by Eric Young
   * (eay@cryptsoft.com).  This product includes software written by Tim
   * Hudson (tjh@cryptsoft.com).
   *
   */

Original SSLeay License
-----------------------

  /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
   * All rights reserved.
   *
```

(continues on next page)

(continued from previous page)

```
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.  The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the rouines from the library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof) from
 *    the apps directory (application code) you must include an acknowledgement:
 *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * The licence and distribution terms for any publically available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply be
 * copied and put under another distribution licence
 * [including the GNU Public Licence.]
 */
```

C.3.14 expat

The pyexpat extension is built using an included copy of the expat sources unless the build is configured --with-system-expat:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                          and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.15 libffi

The _ctypes extension is built using an included copy of the libffi sources unless the build is configured --with-system-libffi:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.16 zlib

The zlib extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

Copyright (C) 1995-2010 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly Mark Adler
jloup@gzip.org madler@alumni.caltech.edu

APPENDIX

D

COPYRIGHT

Python and this documentation is:

Copyright © 2001-2019 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

See History and License for complete license and permissions information.

BIBLIOGRAPHY

[C99] ISO/IEC 9899:1999. "Programming languages – C." A public draft of this standard is available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

PYTHON MODULE INDEX

_
__builtin__, 1262
__future__, 1279
__main__, 1264
_winreg (Windows), 1372

a

abc, 1271
aepack (Mac), 1422
aetools (Mac), 1421
aetypes (Mac), 1423
aifc, 1033
al (IRIX), 1427
AL (IRIX), 1429
anydbm, 333
applesingle (Mac), 1448
argparse, 462
array, 195
ast, 1325
asynchat, 760
asyncore, 756
atexit, 1274
audioop, 1029
autoGIL (Mac), 1413

b

base64, 840
BaseHTTPServer, 996
Bastion, 1303
bdb, 1215
binascii, 842
binhex, 842
bisect, 193
bsddb, 338
buildtools (Mac), 1448
bz2, 368

c

calendar, 170
Carbon.AE (Mac), 1414
Carbon.AH (Mac), 1414
Carbon.App (Mac), 1414
Carbon.Appearance (Mac), 1414
Carbon.CarbonEvents (Mac), 1416
Carbon.CarbonEvt (Mac), 1416
Carbon.CF (Mac), 1414
Carbon.CG (Mac), 1416
Carbon.Cm (Mac), 1416
Carbon.Components (Mac), 1416
Carbon.ControlAccessor (Mac), 1416
Carbon.Controls (Mac), 1416
Carbon.CoreFounation (Mac), 1416
Carbon.CoreGraphics (Mac), 1416
Carbon.Ctl (Mac), 1416
Carbon.Dialogs (Mac), 1416
Carbon.Dlg (Mac), 1416
Carbon.Drag (Mac), 1416
Carbon.Dragconst (Mac), 1416
Carbon.Events (Mac), 1416
Carbon.Evt (Mac), 1416
Carbon.File (Mac), 1416
Carbon.Files (Mac), 1416
Carbon.Fm (Mac), 1416
Carbon.Folder (Mac), 1416
Carbon.Folders (Mac), 1416
Carbon.Fonts (Mac), 1416
Carbon.Help (Mac), 1416
Carbon.IBCarbon (Mac), 1416
Carbon.IBCarbonRuntime (Mac), 1416
Carbon.Icns (Mac), 1416
Carbon.Icons (Mac), 1416
Carbon.Launch (Mac), 1416
Carbon.LaunchServices (Mac), 1416
Carbon.List (Mac), 1416
Carbon.Lists (Mac), 1416
Carbon.MacHelp (Mac), 1416
Carbon.MediaDescr (Mac), 1416
Carbon.Menu (Mac), 1416
Carbon.Menus (Mac), 1416
Carbon.Mlte (Mac), 1416
Carbon.OSA (Mac), 1416
Carbon.OSAconst (Mac), 1416
Carbon.Qd (Mac), 1416
Carbon.Qdoffs (Mac), 1416

Carbon.QDOffscreen (Mac), 1416
Carbon.Qt (Mac), 1416
Carbon.QuickDraw (Mac), 1416
Carbon.QuickTime (Mac), 1416
Carbon.Res (Mac), 1416
Carbon.Resources (Mac), 1416
Carbon.Scrap (Mac), 1416
Carbon.Snd (Mac), 1417
Carbon.Sound (Mac), 1417
Carbon.TE (Mac), 1417
Carbon.TextEdit (Mac), 1417
Carbon.Win (Mac), 1417
Carbon.Windows (Mac), 1417
cd (IRIX), 1429
cfmfile (Mac), 1448
cgi, 914
CGIHTTPServer, 1001
cgitb, 921
chunk, 1040
cmath, 230
cmd, 1067
code, 1295
codecs, 124
codeop, 1297
collections, 174
ColorPicker (Mac), 1417
colorsys, 1041
commands (Unix), 1401
compileall, 1339
compiler, 1351
compiler.ast, 1352
compiler.visitor, 1357
ConfigParser, 395
contextlib, 1269
Cookie, 1011
cookielib, 1002
copy, 216
copy_reg, 328
cPickle, 328
cProfile, 1227
crypt (Unix), 1386
cStringIO, 121
csv, 387
ctypes, 586
curses (Unix), 554
curses.ascii, 573
curses.panel, 575
curses.textpad, 571

d

datetime, 145
dbhash, 337
dbm (Unix), 335
decimal, 233
DEVICE (IRIX), 1441
difflib, 109
dircache, 314
dis, 1341
distutils, 1243
dl (Unix), 1387
doctest, 1152
DocXMLRPCServer, 1027
dumbdbm, 341
dummy_thread, 639
dummy_threading, 638

e

EasyDialogs (Mac), 1407
email, 765
email.charset, 781
email.encoders, 784
email.errors, 784
email.generator, 775
email.header, 779
email.iterators, 787
email.message, 765
email.mime, 777
email.parser, 772
email.utils, 785
encodings.idna, 138
encodings.utf_8_sig, 139
ensurepip, 1243
errno, 580
exceptions, 67

f

fcntl (Unix), 1390
filecmp, 302
fileinput, 295
findertools (Mac), 1407
fl (IRIX), 1433
FL (IRIX), 1437
flp (IRIX), 1438
fm (IRIX), 1438
fnmatch, 308
formatter, 1359
fpectl (Unix), 1293
fpformat, 142
fractions, 260
FrameWork (Mac), 1410
ftplib, 956
functools, 279
future_builtins, 1263

g

gc, 1281
gdbm (Unix), 335
gensuitemodule (Mac), 1420

getopt, 519
getpass, 553
gettext, 1049
gl (IRIX), 1439
GL (IRIX), 1441
glob, 307
grp (Unix), 1385
gzip, 366

h

hashlib, 409
heapq, 189
hmac, 411
hotshot, 1233
hotshot.stats, 1234
htmlentitydefs, 856
htmllib, 855
HTMLParser, 847
httplib, 949

i

ic (Mac), 1403
icopen (Mac), 1448
imageop, 1032
imaplib, 962
imgfile (IRIX), 1441
imghdr, 1042
imp, 1305
importlib, 1308
imputil, 1309
inspect, 1283
io, 445
itertools, 266

j

jpeg (IRIX), 1442
json, 797

k

keyword, 1334

l

lib2to3, 1207
linecache, 309
locale, 1059
logging, 521
logging.config, 535
logging.handlers, 545

m

macerrors (Mac), 1448
MacOS (Mac), 1404
macostools (Mac), 1406
macpath, 315
macresource (Mac), 1448
mailbox, 806
mailcap, 805
marshal, 331
math, 226
md5, 413
mhlib, 826
mimetools, 828
mimetypes, 829
MimeWriter, 832
mimify, 833
MiniAEFrame (Mac), 1424
mmap, 692
modulefinder, 1316
msilib (Windows), 1365
msvcrt (Windows), 1370
multifile, 834
multiprocessing, 639
multiprocessing.connection, 665
multiprocessing.dummy, 669
multiprocessing.managers, 657
multiprocessing.pool, 663
multiprocessing.sharedctypes, 655
mutex, 203

n

Nav (Mac), 1448
netrc, 403
new, 216
nis (Unix), 1399
nntplib, 968
numbers, 223

o

operator, 282
optparse, 492
os, 415
os.path, 291
ossaudiodev (Linux, FreeBSD), 1044

p

parser, 1321
pdb, 1219
pickle, 317
pickletools, 1350
pipes (Unix), 1392
PixMapWrapper (Mac), 1448
pkgutil, 1314
platform, 576
plistlib, 407
popen2, 754
poplib, 960
posix (Unix), 1383
posixfile (Unix), 1394

pprint, 217
profile, 1227
pstats, 1228
pty (Linux), 1390
pwd (Unix), 1384
py_compile, 1339
pyclbr, 1337
pydoc, 1151

q

Queue, 204
quopri, 844

r

random, 262
re, 88
readline (Unix), 695
resource (Unix), 1396
rexec, 1299
rfc822, 836
rlcompleter, 699
robotparser, 402
runpy, 1318

s

sched, 201
ScrolledText (Tk), 1109
select, 621
sets, 198
sgmllib, 852
sha, 414
shelve, 329
shlex, 1069
shutil, 309
signal, 751
SimpleHTTPServer, 1000
SimpleXMLRPCServer, 1023
site, 1289
smtpd, 977
smtplib, 972
sndhdr, 1043
socket, 713
SocketServer, 988
spwd (Unix), 1385
sqlite3, 342
ssl, 726
stat, 297
statvfs, 301
string, 75
StringIO, 120
stringprep, 141
struct, 104
subprocess, 701
sunau, 1035
sunaudiodev (SunOS), 1445
SUNAUDIODEV (SunOS), 1446
symbol, 1332
symtable, 1330
sys, 1247
sysconfig, 1259
syslog (Unix), 1399

t

tabnanny, 1337
tarfile, 376
telnetlib, 978
tempfile, 304
termios (Unix), 1388
test, 1207
test.support, 1210
textwrap, 122
thread, 637
threading, 626
time, 456
timeit, 1234
Tix, 1104
Tkinter, 1075
token, 1333
tokenize, 1335
trace, 1239
traceback, 1276
ttk, 1086
tty (Unix), 1389
turtle, 1110
types, 213

u

unicodedata, 139
unittest, 1177
urllib, 930
urllib2, 937
urlparse, 984
user, 1292
UserDict, 210
UserList, 211
UserString, 212
uu, 845
uuid, 981

v

videoreader (Mac), 1449

w

W (Mac), 1449
warnings, 1264
wave, 1038
weakref, 206
webbrowser, 911

whichdb, 334
winsound (Windows), 1380
wsgiref, 921
wsgiref.handlers, 927
wsgiref.headers, 924
wsgiref.simple_server, 925
wsgiref.util, 922
wsgiref.validate, 926

X

xdrlib, 404
xml, 857
xml.dom, 872
xml.dom.minidom, 883
xml.dom.pulldom, 887
xml.etree.ElementTree, 859
xml.parsers.expat, 900
xml.sax, 888
xml.sax.handler, 890
xml.sax.saxutils, 895
xml.sax.xmlreader, 896
xmlrpclib, 1015

Z

zipfile, 370
zipimport, 1312
zlib, 363

INDEX

Symbols

!=
operator, 32
*
operator, 33
**
operator, 33
+
operator, 33
-
operator, 33
–help
trace command line option, 1239
–ignore-dir=<dir>
trace command line option, 1240
–ignore-module=<mod>
trace command line option, 1240
–user-base
site command line option, 1291
–user-site
site command line option, 1292
–version
trace command line option, 1239
-C, –coverdir=<dir>
trace command line option, 1240
-R, –no-report
trace command line option, 1240
-T, –trackcalls
trace command line option, 1239
-b, –buffer
unittest command line option, 1180
-c <zipfile> <source1> ... <sourceN>
zipfile command line option, 376
-c, –catch
unittest command line option, 1180
-c, –clock
timeit command line option, 1237
-c, –count
trace command line option, 1239
-d destdir
compileall command line option, 1340
-e <zipfile> <output_dir>
zipfile command line option, 376
-f
compileall command line option, 1340
-f, –failfast
unittest command line option, 1180
-f, –file=<file>
trace command line option, 1240
-g, –timing
trace command line option, 1240
-h, –help
timeit command line option, 1237
-i list
compileall command line option, 1340
-l
compileall command line option, 1340
-l <zipfile>
zipfile command line option, 376
-l, –listfuncs
trace command line option, 1239
-m, –missing
trace command line option, 1240
-n N, –number=N
timeit command line option, 1237
-p, –pattern pattern
unittest-discover command line option, 1180
-q
compileall command line option, 1340
-r N, –repeat=N
timeit command line option, 1237
-r, –report
trace command line option, 1239
-s S, –setup=S
timeit command line option, 1237
-s, –start-directory directory
unittest-discover command line option, 1180
-s, –summary
trace command line option, 1240
-t <zipfile>
zipfile command line option, 376
-t, –time
timeit command line option, 1237
-t, –top-level-directory directory

unittest-discover command line option, 1180
-t, –trace
trace command line option, 1239
-v, –verbose
timeit command line option, 1237
unittest-discover command line option, 1180
-x regex
compileall command line option, 1340
..., 1451
.ini
file, 395
.pdbrc
file, 1222
.pythonrc.py
file, 1292
/
operator, 33
//
operator, 33
==
operator, 32
%
operator, 33
% formatting, 46
% interpolation, 46
&
operator, 35
_CData (class in ctypes), 614
_FuncPtr (class in ctypes), 608
_Pointer (class in ctypes), 619
_SimpleCData (class in ctypes), 615
__abs__() (in module operator), 283
__add__() (in module operator), 283
__add__() (rfc822.AddressList method), 839
__and__() (in module operator), 283
__bases__ (class attribute), 66
__builtin__ (module), 1262
__class__ (instance attribute), 66
__cmp__() (instance method), 32
__concat__() (in module operator), 284
__contains__() (email.message.Message method), 768
__contains__() (in module operator), 284
__contains__() (mailbox.Mailbox method), 808
__copy__() (copy protocol), 217
__debug__ (built-in variable), 29
__deepcopy__() (copy protocol), 217
__del__() (io.IOBase method), 449
__delitem__() (email.message.Message method), 768
__delitem__() (in module operator), 285
__delitem__() (mailbox.MH method), 812
__delitem__() (mailbox.Mailbox method), 807
__delslice__() (in module operator), 285
__dict__ (object attribute), 66
__displayhook__ (in module sys), 1248
__div__() (in module operator), 283
__enter__() (_winreg.PyHKEY method), 1380
__enter__() (contextmanager method), 62
__eq__() (email.charset.Charset method), 783
__eq__() (email.header.Header method), 780
__eq__() (in module operator), 282
__eq__() (instance method), 32
__excepthook__ (in module sys), 1248
__exit__() (_winreg.PyHKEY method), 1380
__exit__() (contextmanager method), 63
__floordiv__() (in module operator), 283
__format__, 11
__format__() (datetime.date method), 151
__format__() (datetime.datetime method), 157
__format__() (datetime.time method), 160
__future__, 1454
__future__ (module), 1279
__ge__() (in module operator), 282
__ge__() (instance method), 32
__getinitargs__() (object method), 322
__getitem__() (email.message.Message method), 768
__getitem__() (in module operator), 285
__getitem__() (mailbox.Mailbox method), 808
__getnewargs__() (object method), 322
__getslice__() (in module operator), 285
__getstate__() (object method), 322
__gt__() (in module operator), 282
__gt__() (instance method), 32
__iadd__() (in module operator), 285
__iadd__() (rfc822.AddressList method), 839
__iand__() (in module operator), 286
__iconcat__() (in module operator), 286
__idiv__() (in module operator), 286
__ifloordiv__() (in module operator), 286
__ilshift__() (in module operator), 286
__imod__() (in module operator), 286
__import__() (built-in function), 24
__imul__() (in module operator), 286
__index__() (in module operator), 283
__init__() (logging.Handler method), 525
__inv__() (in module operator), 283
__invert__() (in module operator), 283
__ior__() (in module operator), 286
__ipow__() (in module operator), 286
__irepeat__() (in module operator), 286
__irshift__() (in module operator), 287
__isub__() (in module operator), 287
__isub__() (rfc822.AddressList method), 839
__iter__() (container method), 37
__iter__() (iterator method), 37
__iter__() (mailbox.Mailbox method), 807

__iter__() (unittest.TestSuite method), 1194
__itruediv__() (in module operator), 287
__ixor__() (in module operator), 287
__le__() (in module operator), 282
__le__() (instance method), 32
__len__() (email.message.Message method), 767
__len__() (mailbox.Mailbox method), 808
__len__() (rfc822.AddressList method), 839
__lshift__() (in module operator), 284
__lt__() (in module operator), 282
__lt__() (instance method), 32
__main__
    module, 1318, 1319
__main__ (module), 1264
__members__ (object attribute), 66
__methods__ (object attribute), 66
__missing__(), 53
__missing__() (collections.defaultdict method), 180
__mod__() (in module operator), 284
__mro__ (class attribute), 66
__mul__() (in module operator), 284
__name__ (definition attribute), 66
__ne__() (email.charset.Charset method), 783
__ne__() (email.header.Header method), 780
__ne__() (in module operator), 282
__ne__() (instance method), 32
__neg__() (in module operator), 284
__not__() (in module operator), 283
__or__() (in module operator), 284
__pos__() (in module operator), 284
__pow__() (in module operator), 284
__reduce__() (object method), 323
__reduce_ex__() (object method), 323
__repeat__() (in module operator), 285
__repr__() (multiprocessing.managers.BaseProxy method), 663
__repr__() (netrc.netrc method), 403
__rshift__() (in module operator), 284
__setitem__() (email.message.Message method), 768
__setitem__() (in module operator), 285
__setitem__() (mailbox.Mailbox method), 807
__setitem__() (mailbox.Maildir method), 810
__setslice__() (in module operator), 285
__setstate__() (object method), 322
__slots__, 1458
__stderr__ (in module sys), 1258
__stdin__ (in module sys), 1258
__stdout__ (in module sys), 1258
__str__() (datetime.date method), 151
__str__() (datetime.datetime method), 157
__str__() (datetime.time method), 160
__str__() (email.charset.Charset method), 783
__str__() (email.header.Header method), 780
__str__() (email.message.Message method), 766
__str__() (multiprocessing.managers.BaseProxy method), 663
__str__() (rfc822.AddressList method), 839
__sub__() (in module operator), 284
__sub__() (rfc822.AddressList method), 839
__subclasses__() (class method), 66
__subclasshook__() (abc.ABCMeta method), 1272
__truediv__() (in module operator), 284
__unicode__() (email.header.Header method), 780
__xor__() (in module operator), 284
anonymous (ctypes.Structure attribute), 618
_asdict() (collections.somenamedtuple method), 183
_b_base_ (ctypes._CData attribute), 615
_b_needsfree_ (ctypes._CData attribute), 615
_callmethod() (multiprocessing.managers.BaseProxy method), 662
_clear_type_cache() (in module sys), 1247
_current_frames() (in module sys), 1247
_exit() (in module os), 436
_fields (ast.AST attribute), 1326
_fields (collections.somenamedtuple attribute), 184
fields (ctypes.Structure attribute), 618
_flush() (wsgiref.handlers.BaseHandler method), 928
_getframe() (in module sys), 1252
_getvalue() (multiprocessing.managers.BaseProxy method), 663
_handle (ctypes.PyDLL attribute), 607
_https_verify_certificates() (in module ssl), 729
length (ctypes.Array attribute), 619
_locale
    module, 1059
_make() (collections.somenamedtuple class method), 183
_makeResult() (unittest.TextTestRunner method), 1198
_name (ctypes.PyDLL attribute), 607
_objects (ctypes._CData attribute), 615
pack (ctypes.Structure attribute), 618
_parse() (gettext.NullTranslations method), 1052
_quit() (FrameWork.Application method), 1412
_replace() (collections.somenamedtuple method), 184
_setroot() (xml.etree.ElementTree.ElementTree method), 869
_start() (aetools.TalkTo method), 1421
_structure() (in module email.iterators), 787
type (ctypes.Array attribute), 619
type (ctypes._Pointer attribute), 619
_urlopener (in module urllib), 933
_winreg (module), 1372
_write() (wsgiref.handlers.BaseHandler method), 928

^
    operator, 35
~
    operator, 35
|
    operator, 35
>>>, 1451
>
    operator, 32
>=
    operator, 32
>>
    operator, 35
<
    operator, 32
<=
    operator, 32
<<
    operator, 35
<protocol>_proxy, 940
2to3, 1451

A

A-LAW, 1035, 1043
a-LAW, 1029
a2b_base64() (in module binascii), 843
a2b_hex() (in module binascii), 844
a2b_hqx() (in module binascii), 843
a2b_qp() (in module binascii), 843
a2b_uu() (in module binascii), 842
abc (module), 1271
ABCMeta (class in abc), 1272
abort() (ftplib.FTP method), 958
abort() (in module os), 435
above() (curses.panel.Panel method), 576
abs() (built-in function), 5
abs() (decimal.Context method), 248
abs() (in module operator), 283
abspath() (in module os.path), 291
abstract base class, 1451
AbstractBasicAuthHandler (class in urllib2), 940
AbstractDigestAuthHandler (class in urllib2), 940
AbstractFormatter (class in formatter), 1361
abstractmethod() (in module abc), 1273
abstractproperty() (in module abc), 1274
AbstractWriter (class in formatter), 1362
accept() (asyncore.dispatcher method), 758
accept() (multiprocessing.connection.Listener method), 666
accept() (socket.socket method), 719
accept2dyear (in module time), 457
access() (in module os), 425
acos() (in module cmath), 232
acos() (in module math), 228
acosh() (in module cmath), 232
acosh() (in module math), 229
acquire() (logging.Handler method), 525
acquire() (multiprocessing.Lock method), 652
acquire() (multiprocessing.RLock method), 653
acquire() (thread.lock method), 638
acquire() (threading.Condition method), 633
acquire() (threading.Lock method), 631
acquire() (threading.RLock method), 632
acquire() (threading.Semaphore method), 634
acquire_lock() (in module imp), 1306
Action (class in argparse), 481
action (optparse.Option attribute), 505
ACTIONS (optparse.Option attribute), 518
activate_form() (fl.form method), 1435
active_children() (in module multiprocessing), 649
active_count() (in module threading), 627
activeCount() (in module threading), 627
add() (decimal.Context method), 248
add() (frozenset method), 52
add() (in module audioop), 1029
add() (in module operator), 283
add() (mailbox.Mailbox method), 807
add() (mailbox.Maildir method), 810
add() (msilib.RadioButtonGroup method), 1369
add() (pstats.Stats method), 1229
add() (tarfile.TarFile method), 381
add() (ttk.Notebook method), 1093
add_alias() (in module email.charset), 783
add_argument() (argparse.ArgumentParser method), 471
add_argument_group() (argparse.ArgumentParser method), 488
add_box() (fl.form method), 1435
add_browser() (fl.form method), 1436
add_button() (fl.form method), 1435
add_cgi_vars() (wsgiref.handlers.BaseHandler method), 928
add_charset() (in module email.charset), 783
add_choice() (fl.form method), 1436
add_clock() (fl.form method), 1435
add_codec() (in module email.charset), 783
add_cookie_header() (cookielib.CookieJar method), 1004
add_counter() (fl.form method), 1436
add_data() (in module msilib), 1366
add_data() (urllib2.Request method), 941
add_dial() (fl.form method), 1435
add_fallback() (gettext.NullTranslations method), 1052
add_file() (msilib.Directory method), 1368
add_flag() (mailbox.MaildirMessage method), 815
add_flag() (mailbox.mboxMessage method), 817
add_flag() (mailbox.MMDFMessage method), 821

add_flowing_data() (formatter.formatter method), 1360
add_folder() (mailbox.Maildir method), 810
add_folder() (mailbox.MH method), 812
add_handler() (urllib2.OpenerDirector method), 942
add_header() (email.message.Message method), 768
add_header() (urllib2.Request method), 941
add_header() (wsgiref.headers.Headers method), 924
add_history() (in module readline), 697
add_hor_rule() (formatter.formatter method), 1360
add_input() (fl.form method), 1436
add_label() (mailbox.BabylMessage method), 820
add_label_data() (formatter.formatter method), 1360
add_lightbutton() (fl.form method), 1435
add_line_break() (formatter.formatter method), 1359
add_literal_data() (formatter.formatter method), 1360
add_menu() (fl.form method), 1436
add_mutually_exclusive_group() (argparse.ArgumentParser method), 489
add_option() (optparse.OptionParser method), 504
add_parent() (urllib2.BaseHandler method), 943
add_password() (urllib2.HTTPPasswordMgr method), 945
add_positioner() (fl.form method), 1436
add_roundbutton() (fl.form method), 1435
add_section() (ConfigParser.RawConfigParser method), 397
add_sequence() (mailbox.MHMessage method), 819
add_slider() (fl.form method), 1435
add_stream() (in module msilib), 1366
add_subparsers() (argparse.ArgumentParser method), 485
add_suffix() (imputil.ImportManager method), 1309
add_tables() (in module msilib), 1366
add_text() (fl.form method), 1435
add_timer() (fl.form method), 1436
add_type() (in module mimetypes), 830
add_unredirected_header() (urllib2.Request method), 941
add_valslider() (fl.form method), 1435
addch() (curses.window method), 561
addCleanup() (unittest.TestCase method), 1193
addcomponent() (turtle.Shape method), 1136
addError() (unittest.TestResult method), 1198
addExpectedFailure() (unittest.TestResult method), 1198
addFailure() (unittest.TestResult method), 1198
addfile() (tarfile.TarFile method), 381
addFilter() (logging.Handler method), 526
addFilter() (logging.Logger method), 524
addHandler() (logging.Logger method), 524
addheader() (MimeWriter.MimeWriter method), 832
addinfo() (hotshot.Profile method), 1233
addLevelName() (in module logging), 533
addnstr() (curses.window method), 561
AddPackagePath() (in module modulefinder), 1317
address (multiprocessing.connection.Listener attribute), 666
address (multiprocessing.managers.BaseManager attribute), 658
address_family (SocketServer.BaseServer attribute), 991
address_string() (BaseHTTPServer.BaseHTTPRequestHandler method), 999
AddressList (class in rfc822), 837
addresslist (rfc822.AddressList attribute), 840
addressof() (in module ctypes), 611
addshape() (in module turtle), 1134
addsitedir() (in module site), 1291
addSkip() (unittest.TestResult method), 1198
addstr() (curses.window method), 561
addSuccess() (unittest.TestResult method), 1198
addTest() (unittest.TestSuite method), 1194
addTests() (unittest.TestSuite method), 1194
addTypeEqualityFunc() (unittest.TestCase method), 1191
addUnexpectedSuccess() (unittest.TestResult method), 1198
adjusted() (decimal.Decimal method), 239
adler32() (in module zlib), 363
ADPCM, Intel/DVI, 1029
adpcm2lin() (in module audioop), 1029
aepack (module), 1422
AEServer (class in MiniAEFrame), 1424
AEText (class in aetypes), 1423
aetools (module), 1421
aetypes (module), 1423
AF_INET (in module socket), 715
AF_INET6 (in module socket), 715
AF_UNIX (in module socket), 715
aifc (module), 1033
aifc() (aifc.aifc method), 1035
AIFF, 1033, 1040
aiff() (aifc.aifc method), 1034
AIFF-C, 1033, 1040
AL
    module, 1427
AL (module), 1429
al (module), 1427
alarm() (in module signal), 752
alaw2lin() (in module audioop), 1029
ALERT_DESCRIPTION_HANDSHAKE_FAILURE (in module ssl), 736
ALERT_DESCRIPTION_INTERNAL_ERROR (in module ssl), 736

algorithms_available (in module hashlib), 410
algorithms_guaranteed (in module hashlib), 410
alignment() (in module ctypes), 611
all() (built-in function), 5
all_errors (in module ftplib), 957
all_features (in module xml.sax.handler), 891
all_properties (in module xml.sax.handler), 892
allocate_lock() (in module thread), 637
allow_reuse_address (SocketServer.BaseServer attribute), 991
allowed_domains() (cookielib.DefaultCookiePolicy method), 1008
alt() (in module curses.ascii), 575
ALT_DIGITS (in module locale), 1062
altsep (in module os), 444
altzone (in module time), 457
ALWAYS_TYPED_ACTIONS (optparse.Option attribute), 518
AMPER (in module token), 1333
AMPEREQUAL (in module token), 1333
anchor_bgn() (htmllib.HTMLParser method), 856
anchor_end() (htmllib.HTMLParser method), 856
and
    operator, 31, 32
and_() (in module operator), 283
annotate() (in module dircache), 314
answer_challenge() (in module multiprocessing.connection), 665
any() (built-in function), 5
anydbm (module), 333
api_version (in module sys), 1259
apop() (poplib.POP3 method), 961
append() (array.array method), 196
append() (collections.deque method), 177
append() (email.header.Header method), 780
append() (imaplib.IMAP4 method), 964
append() (list method), 48
append() (msilib.CAB method), 1368
append() (pipes.Template method), 1393
append() (xml.etree.ElementTree.Element method), 868
appendChild() (xml.dom.Node method), 876
appendleft() (collections.deque method), 177
AppleEvents, 1407, 1424
applesingle (module), 1448
Application() (in module FrameWork), 1410
application_uri() (in module wsgiref.util), 922
apply (2to3 fixer), 1204
apply() (built-in function), 27
apply() (multiprocessing.pool.multiprocessing.Pool method), 663
apply_async() (multiprocessing.pool.multiprocessing.Pool method), 664
architecture() (in module platform), 576
archive (zipimport.zipimporter attribute), 1313
aRepr (in module repr), 221
argparse (module), 462
args (exceptions.BaseException attribute), 67
args (functools.partial attribute), 282
argtypes (ctypes._FuncPtr attribute), 608
argument, 1451
ArgumentDefaultsHelpFormatter (class in argparse), 467
ArgumentError, 609
ArgumentParser (class in argparse), 464
argv (in module sys), 1247
arithmetic, 33
ArithmeticError, 67
array (class in array), 196
Array (class in ctypes), 619
array (module), 195
Array() (in module multiprocessing), 655
Array() (in module multiprocessing.sharedctypes), 655
Array() (multiprocessing.managers.SyncManager method), 659
arrays, 195
ArrayType (in module array), 196
article() (nntplib.NNTP method), 971
as_integer_ratio() (float method), 36
AS_IS (in module formatter), 1359
as_string() (email.message.Message method), 766
as_tuple() (decimal.Decimal method), 239
ascii() (in module curses.ascii), 575
ascii() (in module future_builtins), 1263
ascii_letters (in module string), 75
ascii_lowercase (in module string), 75
ascii_uppercase (in module string), 75
asctime() (in module time), 457
asin() (in module cmath), 232
asin() (in module math), 228
asinh() (in module cmath), 232
asinh() (in module math), 229
AskFileForOpen() (in module EasyDialogs), 1408
AskFileForSave() (in module EasyDialogs), 1409
AskFolder() (in module EasyDialogs), 1409
AskPassword() (in module EasyDialogs), 1408
AskString() (in module EasyDialogs), 1408
AskYesNoCancel() (in module EasyDialogs), 1408
assert
    statement, 68
assert_line_data() (formatter.formatter method), 1361
assertAlmostEqual() (unittest.TestCase method), 1190
assertDictContainsSubset() (unittest.TestCase method), 1191
assertDictEqual() (unittest.TestCase method), 1192

assertEqual() (unittest.TestCase method), 1188
assertFalse() (unittest.TestCase method), 1188
assertGreater() (unittest.TestCase method), 1190
assertGreaterEqual() (unittest.TestCase method), 1190
assertIn() (unittest.TestCase method), 1188
AssertionError, 68
assertIs() (unittest.TestCase method), 1188
assertIsInstance() (unittest.TestCase method), 1189
assertIsNone() (unittest.TestCase method), 1188
assertIsNot() (unittest.TestCase method), 1188
assertIsNotNone() (unittest.TestCase method), 1188
assertItemsEqual() (unittest.TestCase method), 1191
assertLess() (unittest.TestCase method), 1190
assertLessEqual() (unittest.TestCase method), 1190
assertListEqual() (unittest.TestCase method), 1192
assertMultiLineEqual() (unittest.TestCase method), 1191
assertNotAlmostEqual() (unittest.TestCase method), 1190
assertNotEqual() (unittest.TestCase method), 1188
assertNotIn() (unittest.TestCase method), 1188
assertNotIsInstance() (unittest.TestCase method), 1189
assertNotRegexpMatches() (unittest.TestCase method), 1190
assertRaises() (unittest.TestCase method), 1189
assertRaisesRegexp() (unittest.TestCase method), 1189
assertRegexpMatches() (unittest.TestCase method), 1190
asserts (2to3 fixer), 1204
assertSequenceEqual() (unittest.TestCase method), 1191
assertSetEqual() (unittest.TestCase method), 1192
assertTrue() (unittest.TestCase method), 1188
assertTupleEqual() (unittest.TestCase method), 1192
assignment
  extended slice, 48
  slice, 48
  subscript, 48
AST (class in ast), 1325
ast (module), 1325
astimezone() (datetime.datetime method), 155
ASTVisitor (class in compiler.visitor), 1357
async_chat (class in asynchat), 760
async_chat.ac_in_buffer_size (in module asynchat), 760
async_chat.ac_out_buffer_size (in module asynchat), 760
asyncevents() (FrameWork.Application method), 1411
asynchat (module), 760
asyncore (module), 756
AsyncResult (class in multiprocessing.pool), 664
AT (in module token), 1333
atan() (in module cmath), 232
atan() (in module math), 229
atan2() (in module math), 229
atanh() (in module cmath), 232
atanh() (in module math), 229
atexit (module), 1274
atime (in module cd), 1430
atof() (in module locale), 1063
atof() (in module string), 85
atoi() (in module locale), 1064
atoi() (in module string), 86
atol() (in module string), 86
attach() (email.message.Message method), 766
AttlistDeclHandler() (xml.parsers.expat.xmlparser method), 904
attrgetter() (in module operator), 287
attrib (xml.etree.ElementTree.Element attribute), 867
attribute, 1451
AttributeError, 68
attributes (xml.dom.Node attribute), 875
AttributesImpl (class in xml.sax.xmlreader), 897
AttributesNSImpl (class in xml.sax.xmlreader), 897
attroff() (curses.window method), 561
attron() (curses.window method), 561
attrset() (curses.window method), 561
audio (in module cd), 1430
Audio Interchange File Format, 1033, 1040
AUDIO_FILE_ENCODING_ADPCM_G721 (in module sunau), 1036
AUDIO_FILE_ENCODING_ADPCM_G722 (in module sunau), 1036
AUDIO_FILE_ENCODING_ADPCM_G723_3 (in module sunau), 1036
AUDIO_FILE_ENCODING_ADPCM_G723_5 (in module sunau), 1036
AUDIO_FILE_ENCODING_ALAW_8 (in module sunau), 1036
AUDIO_FILE_ENCODING_DOUBLE (in module sunau), 1036
AUDIO_FILE_ENCODING_FLOAT (in module sunau), 1036
AUDIO_FILE_ENCODING_LINEAR_16 (in module sunau), 1036
AUDIO_FILE_ENCODING_LINEAR_24 (in module sunau), 1036
AUDIO_FILE_ENCODING_LINEAR_32 (in module sunau), 1036
AUDIO_FILE_ENCODING_LINEAR_8 (in module sunau), 1036
AUDIO_FILE_ENCODING_MULAW_8 (in module sunau), 1036

AUDIO_FILE_MAGIC (in module sunau), 1036
AUDIODEV, 1044
audioop (module), 1029
auth() (ftplib.FTP_TLS method), 960
authenticate() (imaplib.IMAP4 method), 964
AuthenticationError, 666
authenticators() (netrc.netrc method), 403
authkey (multiprocessing.Process attribute), 646
autoGIL (module), 1413
AutoGILError, 1413
avg() (in module audioop), 1029
avgpp() (in module audioop), 1029

B

b16decode() (in module base64), 841
b16encode() (in module base64), 841
b2a_base64() (in module binascii), 843
b2a_hex() (in module binascii), 844
b2a_hqx() (in module binascii), 843
b2a_qp() (in module binascii), 843
b2a_uu() (in module binascii), 843
b32decode() (in module base64), 840
b32encode() (in module base64), 840
b64decode() (in module base64), 840
b64encode() (in module base64), 840
Babyl (class in mailbox), 813
BabylMailbox (class in mailbox), 824
BabylMessage (class in mailbox), 819
back() (in module turtle), 1113
BACKQUOTE (in module token), 1333
backslashreplace_errors() (in module codecs), 127
backward() (in module turtle), 1113
backward_compatible (in module imageop), 1033
BadStatusLine, 951
BadZipfile, 371
Balloon (class in Tix), 1105
base64
    encoding, 840
    module, 842
base64 (module), 840
BaseCGIHandler (class in wsgiref.handlers), 927
BaseCookie (class in Cookie), 1011
BaseException, 67
BaseHandler (class in urllib2), 940
BaseHandler (class in wsgiref.handlers), 927
BaseHTTPRequestHandler (class in BaseHTTPServer), 997
BaseHTTPServer (module), 996
BaseManager (class in multiprocessing.managers), 657
basename() (in module os.path), 291
BaseProxy (class in multiprocessing.managers), 662
BaseRequestHandler (class in SocketServer), 992
BaseServer (class in SocketServer), 990
basestring (2to3 fixer), 1204
basestring() (built-in function), 6
basicConfig() (in module logging), 533
BasicContext (class in decimal), 246
Bastion (module), 1303
Bastion() (in module Bastion), 1303
BastionClass (class in Bastion), 1303
baudrate() (in module curses), 555
bbox() (ttk.Treeview method), 1097
bdb
    module, 1219
Bdb (class in bdb), 1216
bdb (module), 1215
BdbQuit, 1215
BDFL, 1452
beep() (in module curses), 555
Beep() (in module winsound), 1380
begin_fill() (in module turtle), 1123
begin_poly() (in module turtle), 1128
below() (curses.panel.Panel method), 576
Benchmarking, 1234
benchmarking, 457
betavariate() (in module random), 264
bgcolor() (in module turtle), 1130
bgn_group() (fl.form method), 1435
bgpic() (in module turtle), 1130
bias() (in module audioop), 1029
bidirectional() (in module unicodedata), 140
BigEndianStructure (class in ctypes), 618
bin() (built-in function), 6
binary
    data, packing, 104
Binary (class in msilib), 1366
Binary (class in xmlrpclib), 1019
binary semaphores, 637
BINARY_ADD (opcode), 1344
BINARY_AND (opcode), 1344
BINARY_DIVIDE (opcode), 1343
BINARY_FLOOR_DIVIDE (opcode), 1343
BINARY_LSHIFT (opcode), 1344
BINARY_MODULO (opcode), 1343
BINARY_MULTIPLY (opcode), 1343
BINARY_OR (opcode), 1344
BINARY_POWER (opcode), 1343
BINARY_RSHIFT (opcode), 1344
BINARY_SUBSCR (opcode), 1344
BINARY_SUBTRACT (opcode), 1344
BINARY_TRUE_DIVIDE (opcode), 1343
BINARY_XOR (opcode), 1344
binascii (module), 842
bind (widgets), 1084
bind() (asyncore.dispatcher method), 758
bind() (socket.socket method), 719

bind_textdomain_codeset() (in module gettext), 1049
bindtextdomain() (in module gettext), 1049
bindtextdomain() (in module locale), 1065
binhex
    module, 842
binhex (module), 842
binhex() (in module binhex), 842
bisect (module), 193
bisect() (in module bisect), 193
bisect_left() (in module bisect), 193
bisect_right() (in module bisect), 193
bit_length() (int method), 35
bit_length() (long method), 35
bitmap() (msilib.Dialog method), 1370
bitwise
    operations, 35
bk() (in module turtle), 1113
bkgd() (curses.window method), 561
bkgdset() (curses.window method), 562
blocked_domains() (cookielib.DefaultCookiePolicy method), 1008
BlockingIOError, 447
BLOCKSIZE (in module cd), 1430
blocksize (in module sha), 414
body() (nntplib.NNTP method), 971
body_encode() (email.charset.Charset method), 783
body_encoding (email.charset.Charset attribute), 781
body_line_iterator() (in module email.iterators), 787
BOM (in module codecs), 128
BOM_BE (in module codecs), 128
BOM_LE (in module codecs), 128
BOM_UTF16 (in module codecs), 128
BOM_UTF16_BE (in module codecs), 128
BOM_UTF16_LE (in module codecs), 128
BOM_UTF32 (in module codecs), 128
BOM_UTF32_BE (in module codecs), 128
BOM_UTF32_LE (in module codecs), 128
BOM_UTF8 (in module codecs), 128
bool (built-in class), 6
Boolean
    object, 33
    operations, 31, 32
    type, 6
    values, 65
Boolean (class in aetypes), 1423
boolean() (in module xmlrpclib), 1022
BooleanType (in module types), 214
bootstrap() (in module ensurepip), 1245
border() (curses.window method), 562
bottom() (curses.panel.Panel method), 576
bottom_panel() (in module curses.panel), 575
BoundaryError, 784
BoundedSemaphore (class in multiprocessing), 652
BoundedSemaphore() (in module threading), 628
BoundedSemaphore() (multiprocessing.managers.SyncManager method), 658
box() (curses.window method), 562
break_anywhere() (bdb.Bdb method), 1217
break_here() (bdb.Bdb method), 1217
break_long_words (textwrap.TextWrapper attribute), 124
BREAK_LOOP (opcode), 1346
break_on_hyphens (textwrap.TextWrapper attribute), 124
Breakpoint (class in bdb), 1215
breakpoints, 1144
BROWSER, 911, 912
bsddb
    module, 330, 333, 337
bsddb (module), 338
BsdDbShelf (class in shelve), 330
btopen() (in module bsddb), 339
buffer
    built-in function, 215
    object, 37
buffer (2to3 fixer), 1204
buffer (io.TextIOBase attribute), 453
buffer (unittest.TestResult attribute), 1197
buffer size, I/O, 15
buffer() (built-in function), 27
buffer_info() (array.array method), 196
buffer_size (xml.parsers.expat.xmlparser attribute), 902
buffer_text (xml.parsers.expat.xmlparser attribute), 902
buffer_used (xml.parsers.expat.xmlparser attribute), 902
BufferedIOBase (class in io), 449
BufferedRandom (class in io), 452
BufferedReader (class in io), 451
BufferedRWPair (class in io), 452
BufferedWriter (class in io), 452
BufferError, 68
BufferingHandler (class in logging.handlers), 552
BufferTooShort, 646, 666
BufferType (in module types), 215
BUFSIZ (in module macostools), 1407
bufsize() (ossaudiodev.oss_audio_device method), 1046
BUILD_CLASS (opcode), 1346
BUILD_LIST (opcode), 1347
BUILD_MAP (opcode), 1347
build_opener() (in module urllib2), 938
BUILD_SET (opcode), 1347
BUILD_SLICE (opcode), 1349

BUILD_TUPLE (opcode), 1347
buildtools (module), 1448
built-in
    types, 31
built-in function
    buffer, 215
    cmp, 1063
    compile, 65, 214, 1323
    complex, 33
    eval, 65, 86, 219, 1323
    execfile, 1292
    file, 57
    float, 33, 85
    input, 1258
    int, 33
    len, 38, 53
    long, 33, 86
    max, 38
    min, 38
    raw_input, 1258
    reload, 1255, 1306, 1308
    slice, 215, 1349
    type, 65, 214
    xrange, 215
builtin_module_names (in module sys), 1247
BuiltinFunctionType (in module types), 215
BuiltinImporter (class in imputil), 1310
BuiltinMethodType (in module types), 215
ButtonBox (class in Tix), 1105
bye() (in module turtle), 1135
byref() (in module ctypes), 611
byte-code
    file, 1305, 1307, 1339
bytearray
    object, 37
bytearray (built-in class), 6
bytecode, 1452
byteorder (in module sys), 1247
bytes (uuid.UUID attribute), 981
bytes-like object, 1452
bytes_le (uuid.UUID attribute), 981
BytesIO (class in io), 451
byteswap() (array.array method), 196
bz2 (module), 368
BZ2Compressor (class in bz2), 370
BZ2Decompressor (class in bz2), 370
BZ2File (class in bz2), 368

C

C
    language, 33
    structures, 104
c_bool (class in ctypes), 617
C_BUILTIN (in module imp), 1307
c_byte (class in ctypes), 615
c_char (class in ctypes), 616
c_char_p (class in ctypes), 616
c_double (class in ctypes), 616
C_EXTENSION (in module imp), 1306
c_float (class in ctypes), 616
c_int (class in ctypes), 616
c_int16 (class in ctypes), 616
c_int32 (class in ctypes), 616
c_int64 (class in ctypes), 616
c_int8 (class in ctypes), 616
c_long (class in ctypes), 616
c_longdouble (class in ctypes), 616
c_longlong (class in ctypes), 616
c_short (class in ctypes), 616
c_size_t (class in ctypes), 616
c_ssize_t (class in ctypes), 616
c_ubyte (class in ctypes), 616
c_uint (class in ctypes), 617
c_uint16 (class in ctypes), 617
c_uint32 (class in ctypes), 617
c_uint64 (class in ctypes), 617
c_uint8 (class in ctypes), 617
c_ulong (class in ctypes), 617
c_ulonglong (class in ctypes), 617
c_ushort (class in ctypes), 617
c_void_p (class in ctypes), 617
c_wchar (class in ctypes), 617
c_wchar_p (class in ctypes), 617
CAB (class in msilib), 1368
CacheFTPHandler (class in urllib2), 941
calcsize() (in module struct), 105
Calendar (class in calendar), 170
calendar (module), 170
calendar() (in module calendar), 173
call() (dl.dl method), 1388
call() (in module subprocess), 701
CALL_FUNCTION (opcode), 1349
CALL_FUNCTION_KW (opcode), 1350
CALL_FUNCTION_VAR (opcode), 1349
CALL_FUNCTION_VAR_KW (opcode), 1350
call_tracing() (in module sys), 1247
Callable (class in collections), 187
callable() (built-in function), 6
CallableProxyType (in module weakref), 209
callback (optparse.Option attribute), 506
callback() (MiniAEFrame.AEServer method), 1425
callback_args (optparse.Option attribute), 506
callback_kwargs (optparse.Option attribute), 506
CalledProcessError, 703
can_change_color() (in module curses), 555
can_fetch() (robotparser.RobotFileParser method), 402
cancel() (sched.scheduler method), 203

cancel() (threading.Timer method), 636
cancel_join_thread() (multiprocessing.Queue method), 649
CannotSendHeader, 951
CannotSendRequest, 951
canonic() (bdb.Bdb method), 1216
canonical() (decimal.Context method), 248
canonical() (decimal.Decimal method), 239
capitalize() (in module string), 86
capitalize() (str method), 39
captured_stdout() (in module test.support), 1212
captureWarnings() (in module logging), 534
capwords() (in module string), 85
Carbon.AE (module), 1414
Carbon.AH (module), 1414
Carbon.App (module), 1414
Carbon.Appearance (module), 1414
Carbon.CarbonEvents (module), 1416
Carbon.CarbonEvt (module), 1416
Carbon.CF (module), 1414
Carbon.CG (module), 1416
Carbon.Cm (module), 1416
Carbon.Components (module), 1416
Carbon.ControlAccessor (module), 1416
Carbon.Controls (module), 1416
Carbon.CoreFounation (module), 1416
Carbon.CoreGraphics (module), 1416
Carbon.Ctl (module), 1416
Carbon.Dialogs (module), 1416
Carbon.Dlg (module), 1416
Carbon.Drag (module), 1416
Carbon.Dragconst (module), 1416
Carbon.Events (module), 1416
Carbon.Evt (module), 1416
Carbon.File (module), 1416
Carbon.Files (module), 1416
Carbon.Fm (module), 1416
Carbon.Folder (module), 1416
Carbon.Folders (module), 1416
Carbon.Fonts (module), 1416
Carbon.Help (module), 1416
Carbon.IBCarbon (module), 1416
Carbon.IBCarbonRuntime (module), 1416
Carbon.Icns (module), 1416
Carbon.Icons (module), 1416
Carbon.Launch (module), 1416
Carbon.LaunchServices (module), 1416
Carbon.List (module), 1416
Carbon.Lists (module), 1416
Carbon.MacHelp (module), 1416
Carbon.MediaDescr (module), 1416
Carbon.Menu (module), 1416
Carbon.Menus (module), 1416
Carbon.Mlte (module), 1416
Carbon.OSA (module), 1416
Carbon.OSAconst (module), 1416
Carbon.Qd (module), 1416
Carbon.Qdoffs (module), 1416
Carbon.QDOffscreen (module), 1416
Carbon.Qt (module), 1416
Carbon.QuickDraw (module), 1416
Carbon.QuickTime (module), 1416
Carbon.Res (module), 1416
Carbon.Resources (module), 1416
Carbon.Scrap (module), 1416
Carbon.Snd (module), 1417
Carbon.Sound (module), 1417
Carbon.TE (module), 1417
Carbon.TextEdit (module), 1417
Carbon.Win (module), 1417
Carbon.Windows (module), 1417
cast() (in module ctypes), 612
cat() (in module nis), 1399
catalog (in module cd), 1430
catch_warnings (class in warnings), 1269
category() (in module unicodedata), 140
cbreak() (in module curses), 555
cd (module), 1429
CDLL (class in ctypes), 606
CDROM (in module cd), 1430
ceil() (in module math), 34, 226
center() (in module string), 88
center() (str method), 39
CERT_NONE (in module ssl), 732
CERT_OPTIONAL (in module ssl), 732
CERT_REQUIRED (in module ssl), 732
cert_store_stats() (ssl.SSLContext method), 739
cert_time_to_seconds() (in module ssl), 731
CertificateError, 727
certificates, 743
cfmfile (module), 1448
CFUNCTYPE() (in module ctypes), 609
CGI
    debugging, 919
    exceptions, 921
    protocol, 914
    security, 918
    tracebacks, 921
cgi (module), 914
cgi_directories (CGIHTTPServer.CGIHTTPRequestHandler attribute), 1002
CGIHandler (class in wsgiref.handlers), 927
CGIHTTPRequestHandler (class in CGIHTTPServer), 1001
CGIHTTPServer
    module, 996
CGIHTTPServer (module), 1001
cgitb (module), 921

CGIXMLRPCRequestHandler (class in SimpleXMLRPCServer), 1024
chain() (in module itertools), 267
chaining
    comparisons, 32
CHANNEL_BINDING_TYPES (in module ssl), 736
channels() (ossaudiodev.oss_audio_device method), 1046
CHAR_MAX (in module locale), 1064
character, 139
CharacterDataHandler() (xml.parsers.expat.xmlparser method), 904
characters() (xml.sax.handler.ContentHandler method), 893
characters_written (io.BlockingIOError attribute), 447
Charset (class in email.charset), 781
CHARSET (in module mimify), 833
charset() (gettext.NullTranslations method), 1053
chdir() (in module os), 426
check() (imaplib.IMAP4 method), 964
check() (in module tabnanny), 1337
check_call() (in module subprocess), 702
check_forms() (in module fl), 1433
check_hostname (ssl.SSLContext attribute), 743
check_output() (doctest.OutputChecker method), 1173
check_output() (in module subprocess), 702
check_py3k_warnings() (in module test.support), 1212
check_unused_args() (string.Formatter method), 77
check_warnings() (in module test.support), 1211
checkbox() (msilib.Dialog method), 1370
checkcache() (in module linecache), 309
checkfuncname() (in module bdb), 1219
CheckList (class in Tix), 1107
checksum
    Cyclic Redundancy Check, 364
    MD5, 413
    SHA, 414
chflags() (in module os), 426
chgat() (curses.window method), 562
childerr (popen2.Popen3 attribute), 755
childNodes (xml.dom.Node attribute), 875
chmod() (in module os), 427
choice() (in module random), 263
choices (optparse.Option attribute), 506
choose_boundary() (in module mimetools), 828
chown() (in module os), 428
chr() (built-in function), 6
chroot() (in module os), 427
Chunk (class in chunk), 1041
chunk (module), 1040
cipher
    DES, 1386
cipher() (ssl.SSLSocket method), 738
circle() (in module turtle), 1116
CIRCUMFLEX (in module token), 1333
CIRCUMFLEXEQUAL (in module token), 1333
Clamped (class in decimal), 251
class, 1452
Class (class in symtable), 1331
Class browser, 1141
classic class, 1452
classmethod() (built-in function), 6
classobj() (in module new), 216
ClassType (in module types), 214
clean() (mailbox.Maildir method), 810
cleandoc() (in module inspect), 1287
Clear Breakpoint, 1144
clear() (collections.deque method), 177
clear() (cookielib.CookieJar method), 1004
clear() (curses.window method), 562
clear() (dict method), 54
clear() (frozenset method), 52
clear() (in module turtle), 1124, 1130
clear() (mailbox.Mailbox method), 808
clear() (threading.Event method), 635
clear() (xml.etree.ElementTree.Element method), 867
clear_all_breaks() (bdb.Bdb method), 1218
clear_all_file_breaks() (bdb.Bdb method), 1218
clear_bpbynumber() (bdb.Bdb method), 1218
clear_break() (bdb.Bdb method), 1218
clear_flags() (decimal.Context method), 247
clear_history() (in module readline), 697
clear_memo() (pickle.Pickler method), 320
clear_session_cookies() (cookielib.CookieJar method), 1005
clearcache() (in module linecache), 309
ClearData() (msilib.Record method), 1368
clearok() (curses.window method), 562
clearscreen() (in module turtle), 1130
clearstamp() (in module turtle), 1117
clearstamps() (in module turtle), 1117
Client() (in module multiprocessing.connection), 665
client_address (BaseHTTPServer.BaseHTTPRequestHandler attribute), 997
clock() (in module time), 457
clone() (email.generator.Generator method), 776
clone() (in module turtle), 1128
clone() (pipes.Template method), 1393
cloneNode() (xml.dom.minidom.Node method), 885
cloneNode() (xml.dom.Node method), 876
Close() (_winreg.PyHKEY method), 1380
close() (aifc.aifc method), 1034, 1035

close() (asyncore.dispatcher method), 758
close() (bsddb.bsddbobject method), 339
close() (bz2.BZ2File method), 369
close() (chunk.Chunk method), 1041
close() (Connection method), 651
close() (dl.dl method), 1388
close() (email.parser.FeedParser method), 773
close() (file method), 57
close() (FrameWork.Window method), 1412
close() (ftplib.FTP method), 960
close() (hotshot.Profile method), 1233
close() (HTMLParser.HTMLParser method), 848
close() (httplib.HTTPConnection method), 953
close() (imaplib.IMAP4 method), 964
close() (in module anydbm), 334
close() (in module dbm), 335
close() (in module dumbdbm), 341
close() (in module fileinput), 296
close() (in module gdbm), 337
close() (in module os), 421
close() (io.IOBase method), 448
close() (logging.FileHandler method), 545
close() (logging.Handler method), 526
close() (logging.handlers.MemoryHandler method), 552
close() (logging.handlers.NTEventLogHandler method), 551
close() (logging.handlers.SocketHandler method), 548
close() (logging.handlers.SysLogHandler method), 550
close() (mailbox.Mailbox method), 809
close() (mailbox.Maildir method), 810
close() (mailbox.MH method), 813
close() (mmap.mmap method), 694
Close() (msilib.View method), 1367
close() (multiprocessing.connection.Listener method), 666
close() (multiprocessing.pool.multiprocessing.Pool method), 664
close() (multiprocessing.Queue method), 648
close() (ossaudiodev.oss_audio_device method), 1045
close() (ossaudiodev.oss_mixer_device method), 1047
close() (select.epoll method), 623
close() (select.kqueue method), 624
close() (sgmllib.SGMLParser method), 853
close() (shelve.Shelf method), 330
close() (socket.socket method), 720
close() (sqlite3.Connection method), 346
close() (StringIO.StringIO method), 120
close() (sunau.AU_read method), 1037
close() (sunau.AU_write method), 1038
close() (tarfile.TarFile method), 382
close() (telnetlib.Telnet method), 979
close() (urllib2.BaseHandler method), 943
close() (wave.Wave_read method), 1039
close() (wave.Wave_write method), 1040
close() (xml.etree.ElementTree.TreeBuilder method), 871
close() (xml.etree.ElementTree.XMLParser method), 871
close() (xml.sax.xmlreader.IncrementalParser method), 898
close() (zipfile.ZipFile method), 372
close_when_done() (asynchat.async_chat method), 761
closed (file attribute), 60
closed (io.IOBase attribute), 448
closed (ossaudiodev.oss_audio_device attribute), 1047
CloseKey() (in module _winreg), 1372
closelog() (in module syslog), 1400
closerange() (in module os), 422
closing() (in module contextlib), 1271
clrtobot() (curses.window method), 562
clrtoeol() (curses.window method), 562
cmath (module), 230
cmd
    module, 1219
Cmd (class in cmd), 1067
cmd (module), 1067
cmd (subprocess.CalledProcessError attribute), 703
cmdloop() (cmd.Cmd method), 1067
cmdqueue (cmd.Cmd attribute), 1069
cmp
    built-in function, 1063
cmp() (built-in function), 7
cmp() (in module filecmp), 302
cmp_op (in module dis), 1342
cmp_to_key() (in module functools), 280
cmpfiles() (in module filecmp), 302
code (module), 1295
code (urllib2.HTTPError attribute), 939
code (xml.parsers.expat.ExpatError attribute), 906
code object, 64, 332
code() (in module new), 216
Codecs, 124
    decode, 124
    encode, 124
codecs (module), 124
coded_value (Cookie.Morsel attribute), 1013
codeop (module), 1297
codepoint2name (in module htmlentitydefs), 857
CODESET (in module locale), 1061
CodeType (in module types), 214
coerce() (built-in function), 27

coercion, 1452
col_offset (ast.AST attribute), 1326
collapse_rfc2231_value() (in module email.utils), 787
collect() (in module gc), 1281
collect_incoming_data() (asynchat.async_chat method), 761
collections (module), 174
COLON (in module token), 1333
color() (in module fl), 1434
color() (in module turtle), 1123
color_content() (in module curses), 555
color_pair() (in module curses), 555
colormode() (in module turtle), 1133
ColorPicker (module), 1417
colorsys (module), 1041
column() (ttk.Treeview method), 1098
COLUMNS, 560
combinations() (in module itertools), 268
combinations_with_replacement() (in module itertools), 269
combine() (datetime.datetime class method), 153
combining() (in module unicodedata), 140
ComboBox (class in Tix), 1105
Combobox (class in ttk), 1091
COMMA (in module token), 1333
command (BaseHTTPServer.BaseHTTPRequestHandler attribute), 997
CommandCompiler (class in codeop), 1297
commands (module), 1401
comment (cookielib.Cookie attribute), 1010
COMMENT (in module tokenize), 1335
comment (zipfile.ZipFile attribute), 374
comment (zipfile.ZipInfo attribute), 375
Comment() (in module xml.etree.ElementTree), 865
comment_url (cookielib.Cookie attribute), 1010
commenters (shlex.shlex attribute), 1071
CommentHandler() (xml.parsers.expat.xmlparser method), 905
commit() (msilib.CAB method), 1368
Commit() (msilib.Database method), 1366
commit() (sqlite3.Connection method), 346
common (filecmp.dircmp attribute), 303
Common Gateway Interface, 914
common_dirs (filecmp.dircmp attribute), 303
common_files (filecmp.dircmp attribute), 303
common_funny (filecmp.dircmp attribute), 303
common_types (in module mimetypes), 830
commonprefix() (in module os.path), 292
communicate() (subprocess.Popen method), 707
compare() (decimal.Context method), 248
compare() (decimal.Decimal method), 239
compare() (difflib.Differ method), 117
compare_digest() (in module hmac), 412
COMPARE_OP (opcode), 1348
compare_signal() (decimal.Context method), 248
compare_signal() (decimal.Decimal method), 239
compare_total() (decimal.Context method), 248
compare_total() (decimal.Decimal method), 239
compare_total_mag() (decimal.Context method), 248
compare_total_mag() (decimal.Decimal method), 240
comparing
    objects, 32
comparison
    operator, 32
Comparison (class in aetypes), 1424
COMPARISON_FLAGS (in module doctest), 1161
comparisons
    chaining, 32
compile
    built-in function, 65, 214, 1323
Compile (class in codeop), 1297
compile() (built-in function), 7
compile() (in module compiler), 1351
compile() (in module py_compile), 1339
compile() (in module re), 93
compile() (parser.ST method), 1324
compile_command() (in module code), 1295
compile_command() (in module codeop), 1297
compile_dir() (in module compileall), 1340
compile_file() (in module compileall), 1340
compile_path() (in module compileall), 1340
compileall (module), 1339
compileall command line option
    -d destdir, 1340
    -f, 1340
    -i list, 1340
    -l, 1340
    -q, 1340
    -x regex, 1340
    directory ..., 1339
    file ..., 1339
compileFile() (in module compiler), 1352
compiler (module), 1351
compiler.ast (module), 1352
compiler.visitor (module), 1357
compilest() (in module parser), 1323
complete() (rlcompleter.Completer method), 700
complete_statement() (in module sqlite3), 345
completedefault() (cmd.Cmd method), 1068
complex
    built-in function, 33
complex (built-in class), 8
Complex (class in numbers), 223
complex number, 1452
complex number

literals, 33
object, 33
ComplexType (in module types), 214
ComponentItem (class in aetypes), 1424
compress() (bz2.BZ2Compressor method), 370
compress() (in module bz2), 370
compress() (in module itertools), 269
compress() (in module jpeg), 1442
compress() (in module zlib), 363
compress() (zlib.Compress method), 365
compress_size (zipfile.ZipInfo attribute), 376
compress_type (zipfile.ZipInfo attribute), 375
compression() (ssl.SSLSocket method), 738
CompressionError, 378
compressobj() (in module zlib), 364
COMSPEC, 441, 705
concat() (in module operator), 284
concatenation
operation, 38
Condition (class in multiprocessing), 652
Condition (class in threading), 633
condition() (msilib.Control method), 1369
Condition() (multiprocessing.managers.SyncManager method), 659
ConfigParser (class in ConfigParser), 396
ConfigParser (module), 395
configuration
file, 395
file, debugger, 1222
file, path, 1290
file, user, 1292
configuration information, 1259
configure() (ttk.Style method), 1101
confstr() (in module os), 443
confstr_names (in module os), 443
conjugate() (complex number method), 34
conjugate() (decimal.Decimal method), 240
conjugate() (numbers.Complex method), 223
connect() (asyncore.dispatcher method), 758
connect() (ftplib.FTP method), 957
connect() (httplib.HTTPConnection method), 953
connect() (in module sqlite3), 344
connect() (multiprocessing.managers.BaseManager method), 658
connect() (smtplib.SMTP method), 974
connect() (socket.socket method), 720
connect_ex() (socket.socket method), 720
Connection (built-in class), 650
Connection (class in sqlite3), 345
connection (sqlite3.Cursor attribute), 353
ConnectRegistry() (in module _winreg), 1372
const (optparse.Option attribute), 506
constructor() (in module copy_reg), 328
container
iteration over, 37
Container (class in collections), 187
contains() (in module operator), 284
content type
MIME, 829
ContentHandler (class in xml.sax.handler), 890
contents (ctypes._Pointer attribute), 620
ContentTooShortError, 936
Context (class in decimal), 246
context (ssl.SSLSocket attribute), 739
context management protocol, 62
context manager, 1452
context manager, 62
context_diff() (in module difflib), 111
contextlib (module), 1269
contextmanager() (in module contextlib), 1270
CONTINUE_LOOP (opcode), 1346
Control (class in msilib), 1369
Control (class in Tix), 1105
control (in module cd), 1430
control() (msilib.Dialog method), 1370
control() (select.kqueue method), 624
controlnames (in module curses.ascii), 575
controls() (ossaudiodev.oss_mixer_device method), 1047
ConversionError, 406
conversions
numeric, 34
convert() (email.charset.Charset method), 782
convert_arg_line_to_args() (argparse.ArgumentParser method), 491
convert_charref() (sgmllib.SGMLParser method), 853
convert_codepoint() (sgmllib.SGMLParser method), 854
convert_entityref() (sgmllib.SGMLParser method), 854
convert_field() (string.Formatter method), 77
Cookie (class in cookielib), 1003
Cookie (module), 1011
CookieError, 1011
CookieJar (class in cookielib), 1003
cookiejar (urllib2.HTTPCookieProcessor attribute), 945
cookielib (module), 1002
CookiePolicy (class in cookielib), 1003
Coordinated Universal Time, 456
Copy, 1144
copy
module, 328
copy (module), 216
copy() (decimal.Context method), 247
copy() (dict method), 54
copy() (frozenset method), 51

copy() (hashlib.hash method), 411
copy() (hmac.HMAC method), 412
copy() (imaplib.IMAP4 method), 964
copy() (in module copy), 216
copy() (in module findertools), 1407
copy() (in module macostools), 1406
copy() (in module multiprocessing.sharedctypes), 656
copy() (in module shutil), 310
copy() (md5.md5 method), 413
copy() (pipes.Template method), 1394
copy() (sha.sha method), 414
copy() (zlib.Compress method), 365
copy() (zlib.Decompress method), 366
copy2() (in module shutil), 310
copy_abs() (decimal.Context method), 248
copy_abs() (decimal.Decimal method), 240
copy_decimal() (decimal.Context method), 247
copy_location() (in module ast), 1329
copy_negate() (decimal.Context method), 248
copy_negate() (decimal.Decimal method), 240
copy_reg (module), 328
copy_sign() (decimal.Context method), 248
copy_sign() (decimal.Decimal method), 240
copybinary() (in module mimetools), 828
copyfile() (in module shutil), 310
copyfileobj() (in module shutil), 310
copying files, 309
copyliteral() (in module mimetools), 828
copymessage() (mhlib.Folder method), 827
copymode() (in module shutil), 310
copyright (built-in variable), 29
copyright (in module sys), 1247
copysign() (in module math), 226
copystat() (in module shutil), 310
copytree() (in module macostools), 1406
copytree() (in module shutil), 310
cos() (in module cmath), 232
cos() (in module math), 229
cosh() (in module cmath), 232
cosh() (in module math), 229
count() (array.array method), 196
count() (collections.deque method), 177
count() (in module itertools), 270
count() (in module string), 86
count() (list method), 48
count() (str method), 40
Counter (class in collections), 174
countOf() (in module operator), 285
countTestCases() (unittest.TestCase method), 1192
countTestCases() (unittest.TestSuite method), 1194
CoverageResults (class in trace), 1241
cPickle
    module, 328
cPickle (module), 328
cProfile (module), 1227
CPU time, 457
cpu_count() (in module multiprocessing), 649
CPython, 1452
CRC (zipfile.ZipInfo attribute), 376
crc32() (in module binascii), 843
crc32() (in module zlib), 364
crc_hqx() (in module binascii), 843
create() (imaplib.IMAP4 method), 964
create_aggregate() (sqlite3.Connection method), 346
create_collation() (sqlite3.Connection method), 347
create_connection() (in module socket), 716
create_decimal() (decimal.Context method), 247
create_decimal_from_float() (decimal.Context method), 248
create_default_context() (in module ssl), 729
create_function() (sqlite3.Connection method), 346
CREATE_NEW_CONSOLE (in module subprocess), 709
CREATE_NEW_PROCESS_GROUP (in module subprocess), 709
create_socket() (asyncore.dispatcher method), 758
create_stats() (profile.Profile method), 1228
create_string_buffer() (in module ctypes), 612
create_system (zipfile.ZipInfo attribute), 375
create_unicode_buffer() (in module ctypes), 612
create_version (zipfile.ZipInfo attribute), 375
createAttribute() (xml.dom.Document method), 878
createAttributeNS() (xml.dom.Document method), 878
createComment() (xml.dom.Document method), 878
createDocument() (xml.dom.DOMImplementation method), 874
createDocumentType() (xml.dom.DOMImplementation method), 874
createElement() (xml.dom.Document method), 877
createElementNS() (xml.dom.Document method), 877
createfilehandler() (Tkinter.Widget.tk method), 1086
CreateKey() (in module _winreg), 1373
CreateKeyEx() (in module _winreg), 1373
createLock() (logging.Handler method), 525
createLock() (logging.NullHandler method), 546
createparser() (in module cd), 1429
createProcessingInstruction() (xml.dom.Document method), 878
CreateRecord() (in module msilib), 1365
createSocket() (logging.handlers.SocketHandler method), 548
createTextNode() (xml.dom.Document method), 877
credits (built-in variable), 29
critical() (in module logging), 532
critical() (logging.Logger method), 524

CRNCYSTR (in module locale), 1061
crop() (in module imageop), 1032
cross() (in module audioop), 1029
crypt
    module, 1384
crypt (module), 1386
crypt() (in module crypt), 1386
crypt(3), 1386
cryptography, 409
cStringIO (module), 121
csv, 387
csv (module), 387
ctermid() (in module os), 416
ctime() (datetime.date method), 151
ctime() (datetime.datetime method), 157
ctime() (in module time), 457
ctrl() (in module curses.ascii), 575
CTRL_BREAK_EVENT (in module signal), 751
CTRL_C_EVENT (in module signal), 751
ctypes (module), 586
curdir (in module os), 444
currency() (in module locale), 1063
current() (ttk.Combobox method), 1091
current_process() (in module multiprocessing), 650
current_thread() (in module threading), 627
CurrentByteIndex (xml.parsers.expat.xmlparser attribute), 903
CurrentColumnNumber (xml.parsers.expat.xmlparser attribute), 903
currentframe() (in module inspect), 1289
CurrentLineNumber (xml.parsers.expat.xmlparser attribute), 903
currentThread() (in module threading), 627
curs_set() (in module curses), 555
curses (module), 554
curses.ascii (module), 573
curses.panel (module), 575
curses.textpad (module), 571
Cursor (class in sqlite3), 351
cursor() (sqlite3.Connection method), 345
cursyncup() (curses.window method), 562
curval (EasyDialogs.ProgressBar attribute), 1409
Cut, 1144
cwd() (ftplib.FTP method), 959
cycle() (in module itertools), 270
Cyclic Redundancy Check, 364

D

D_FMT (in module locale), 1061
D_T_FMT (in module locale), 1061
daemon (multiprocessing.Process attribute), 645
daemon (threading.Thread attribute), 631
data
    packing binary, 104
    tabular, 387
Data (class in plistlib), 408
data (select.kevent attribute), 626
data (UserDict.IterableUserDict attribute), 211
data (UserList.UserList attribute), 212
data (UserString.MutableString attribute), 213
data (xml.dom.Comment attribute), 880
data (xml.dom.ProcessingInstruction attribute), 880
data (xml.dom.Text attribute), 880
data (xmlrpclib.Binary attribute), 1019
data() (xml.etree.ElementTree.TreeBuilder method), 871
database
    Unicode, 139
databases, 341
DatagramHandler (class in logging.handlers), 549
DatagramRequestHandler (class in SocketServer), 992
DATASIZE (in module cd), 1430
date (class in datetime), 149
date() (datetime.datetime method), 155
date() (nntplib.NNTP method), 972
date_time (zipfile.ZipInfo attribute), 375
date_time_string() (BaseHTTPServer.BaseHTTPRequestHandler method), 999
datetime (class in datetime), 152
DateTime (class in xmlrpclib), 1018
datetime (module), 145
day (datetime.date attribute), 150
day (datetime.datetime attribute), 154
day_abbr (in module calendar), 173
day_name (in module calendar), 173
daylight (in module time), 458
Daylight Saving Time, 456
DbfilenameShelf (class in shelve), 330
dbhash
    module, 333
dbhash (module), 337
dbm
    module, 330, 333, 336
dbm (module), 335
dcgettext() (in module locale), 1065
deactivate_form() (fl.form method), 1435
debug (imaplib.IMAP4 attribute), 968
DEBUG (in module re), 94
debug (shlex.shlex attribute), 1072
debug (zipfile.ZipFile attribute), 374
debug() (in module doctest), 1175
debug() (in module logging), 531
debug() (logging.Logger method), 523
debug() (pipes.Template method), 1393
debug() (unittest.TestCase method), 1187
debug() (unittest.TestSuite method), 1194

DEBUG_COLLECTABLE (in module gc), 1283
DEBUG_INSTANCES (in module gc), 1283
DEBUG_LEAK (in module gc), 1283
DEBUG_OBJECTS (in module gc), 1283
DEBUG_SAVEALL (in module gc), 1283
debug_src() (in module doctest), 1175
DEBUG_STATS (in module gc), 1283
DEBUG_UNCOLLECTABLE (in module gc), 1283
debugger, 1143, 1253, 1257
    configuration file, 1222
debugging, 1219
    CGI, 919
DebuggingServer (class in smtpd), 977
DebugRunner (class in doctest), 1175
DebugStr() (in module MacOS), 1405
Decimal (class in decimal), 237
decimal (module), 233
decimal() (in module unicodedata), 139
DecimalException (class in decimal), 252
decode
    Codecs, 124
decode() (codecs.Codec method), 129
decode() (codecs.IncrementalDecoder method), 130
decode() (in module base64), 841
decode() (in module codecs), 124
decode() (in module mimetools), 828
decode() (in module quopri), 844
decode() (in module uu), 845
decode() (json.JSONDecoder method), 802
decode() (str method), 40
decode() (xmlrpclib.Binary method), 1019
decode() (xmlrpclib.DateTime method), 1018
decode_header() (in module email.header), 780
decode_params() (in module email.utils), 787
decode_rfc2231() (in module email.utils), 786
DecodedGenerator (class in email.generator), 776
decodestring() (in module base64), 841
decodestring() (in module quopri), 845
decomposition() (in module unicodedata), 140
decompress() (bz2.BZ2Decompressor method), 370
decompress() (in module bz2), 370
decompress() (in module jpeg), 1442
decompress() (in module zlib), 364
decompress() (zlib.Decompress method), 366
decompressobj() (in module zlib), 365
decorator, 1452
DEDENT (in module token), 1333
dedent() (in module textwrap), 122
deepcopy() (in module copy), 216
def_prog_mode() (in module curses), 555
def_shell_mode() (in module curses), 555
default (optparse.Option attribute), 506
default() (cmd.Cmd method), 1068
default() (compiler.visitor.ASTVisitor method), 1358
default() (json.JSONEncoder method), 803
DEFAULT_BUFFER_SIZE (in module io), 445
default_bufsize (in module xml.dom.pulldom), 888
default_factory (collections.defaultdict attribute), 180
DEFAULT_FORMAT (in module tarfile), 379
default_open() (urllib2.BaseHandler method), 943
default_timer() (in module timeit), 1235
DefaultContext (class in decimal), 246
DefaultCookiePolicy (class in cookielib), 1003
defaultdict (class in collections), 180
DefaultHandler() (xml.parsers.expat.xmlparser method), 905
DefaultHandlerExpand() (xml.parsers.expat.xmlparser method), 905
defaults() (ConfigParser.RawConfigParser method), 397
defaultTestLoader (in module unittest), 1198
defaultTestResult() (unittest.TestCase method), 1193
defects (email.message.Message attribute), 772
defpath (in module os), 444
degrees() (in module math), 229
degrees() (in module turtle), 1119
del
    statement, 48, 53
del_param() (email.message.Message method), 770
delattr() (built-in function), 8
delay() (in module turtle), 1131
delay_output() (in module curses), 555
delayload (cookielib.FileCookieJar attribute), 1005
delch() (curses.window method), 563
dele() (poplib.POP3 method), 962
delete() (ftplib.FTP method), 959
delete() (imaplib.IMAP4 method), 964
delete() (ttk.Treeview method), 1098
DELETE_ATTR (opcode), 1347
DELETE_FAST (opcode), 1349
DELETE_GLOBAL (opcode), 1347
DELETE_NAME (opcode), 1347
DELETE_SLICE+0 (opcode), 1345
DELETE_SLICE+1 (opcode), 1345
DELETE_SLICE+2 (opcode), 1345
DELETE_SLICE+3 (opcode), 1345
DELETE_SUBSCR (opcode), 1345
deleteacl() (imaplib.IMAP4 method), 965
deletefilehandler() (Tkinter.Widget.tk method), 1086
deletefolder() (mhlib.MH method), 826
DeleteKey() (in module _winreg), 1373
DeleteKeyEx() (in module _winreg), 1373
deleteln() (curses.window method), 563
deleteMe() (bdb.Breakpoint method), 1215
DeleteValue() (in module _winreg), 1374

delimiter (csv.Dialect attribute), 391
delitem() (in module operator), 285
deliver_challenge() (in module multiprocessing.connection), 665
delslice() (in module operator), 285
demo_app() (in module wsgiref.simple_server), 925
denominator (numbers.Rational attribute), 224
DeprecationWarning, 72
deque (class in collections), 177
DER_cert_to_PEM_cert() (in module ssl), 731
derwin() (curses.window method), 563
DES
    cipher, 1386
description (sqlite3.Cursor attribute), 353
description() (nntplib.NNTP method), 971
descriptions() (nntplib.NNTP method), 970
descriptor, 1453
descriptor
    file, 58
dest (optparse.Option attribute), 505
Detach() (_winreg.PyHKEY method), 1380
detach() (io.BufferedIOBase method), 450
detach() (io.TextIOBase method), 453
detach() (ttk.Treeview method), 1098
deterministic profiling, 1224
DEVICE (module), 1441
devnull (in module os), 444
dgettext() (in module gettext), 1050
dgettext() (in module locale), 1065
Dialect (class in csv), 390
dialect (csv.csvreader attribute), 392
dialect (csv.csvwriter attribute), 392
Dialog (class in msilib), 1369
DialogWindow() (in module FrameWork), 1411
dict (2to3 fixer), 1205
dict (built-in class), 53
dict() (multiprocessing.managers.SyncManager method), 659
dictConfig() (in module logging.config), 535
dictionary, 1453
dictionary
    object, 53
    type, operations on, 53
dictionary view, 1453
DictionaryType (in module types), 214
DictMixin (class in UserDict), 211
DictProxyType (in module types), 215
DictReader (class in csv), 389
DictType (in module types), 214
DictWriter (class in csv), 389
diff_files (filecmp.dircmp attribute), 303
Differ (class in difflib), 110, 117
difference() (frozenset method), 51
difference_update() (frozenset method), 52
difflib (module), 109
digest() (hashlib.hash method), 410
digest() (hmac.HMAC method), 412
digest() (md5.md5 method), 413
digest() (sha.sha method), 414
digest_size (in module md5), 413
digest_size (in module sha), 414
digit() (in module unicodedata), 139
digits (in module string), 75
dir() (built-in function), 8
dir() (ftplib.FTP method), 959
dircache (module), 314
dircmp (class in filecmp), 302
directory
    changing, 426
    creating, 429
    deleting, 311, 430
    site-packages, 1290
    site-python, 1290
    traversal, 434
    walking, 434
Directory (class in msilib), 1368
directory ...
    compileall command line option, 1339
DirList (class in Tix), 1106
dirname() (in module os.path), 292
DirSelectBox (class in Tix), 1106
DirSelectDialog (class in Tix), 1106
DirTree (class in Tix), 1106
dis (module), 1341
dis() (in module dis), 1341
dis() (in module pickletools), 1350
disable() (bdb.Breakpoint method), 1215
disable() (in module gc), 1281
disable() (in module logging), 533
disable() (profile.Profile method), 1228
disable_interspersed_args() (optparse.OptionParser method), 510
DisableReflectionKey() (in module _winreg), 1377
disassemble() (in module dis), 1342
discard (cookielib.Cookie attribute), 1010
discard() (frozenset method), 52
discard() (mailbox.Mailbox method), 807
discard() (mailbox.MH method), 812
discard_buffers() (asynchat.async_chat method), 761
disco() (in module dis), 1342
discover() (unittest.TestLoader method), 1195
dispatch() (compiler.visitor.ASTVisitor method), 1358
dispatch_call() (bdb.Bdb method), 1217
dispatch_exception() (bdb.Bdb method), 1217
dispatch_line() (bdb.Bdb method), 1216
dispatch_return() (bdb.Bdb method), 1217

dispatcher (class in asyncore), 757
dispatcher_with_send (class in asyncore), 758
displayhook() (in module sys), 1248
dist() (in module platform), 579
distance() (in module turtle), 1119
distb() (in module dis), 1341
distutils (module), 1243
dither2grey2() (in module imageop), 1033
dither2mono() (in module imageop), 1033
div() (in module operator), 283
divide() (decimal.Context method), 248
divide_int() (decimal.Context method), 249
division
    integer, 34
    long integer, 34
DivisionByZero (class in decimal), 252
divmod() (built-in function), 9
divmod() (decimal.Context method), 249
dl (module), 1387
DllCanUnloadNow() (in module ctypes), 612
DllGetClassObject() (in module ctypes), 612
dllhandle (in module sys), 1248
dngettext() (in module gettext), 1050
do_activate() (FrameWork.ScrolledWindow method), 1413
do_activate() (FrameWork.Window method), 1412
do_char() (FrameWork.Application method), 1412
do_clear() (bdb.Bdb method), 1217
do_command() (curses.textpad.Textbox method), 572
do_contentclick() (FrameWork.Window method), 1412
do_controlhit() (FrameWork.ControlsWindow method), 1412
do_controlhit() (FrameWork.ScrolledWindow method), 1413
do_dialogevent() (FrameWork.Application method), 1412
do_forms() (in module fl), 1433
do_GET() (SimpleHTTPServer.SimpleHTTPRequestHandler method), 1000
do_handshake() (ssl.SSLSocket method), 737
do_HEAD() (SimpleHTTPServer.SimpleHTTPRequestHandler method), 1000
do_itemhit() (FrameWork.DialogWindow method), 1413
do_POST() (CGIHTTPServer.CGIHTTPRequestHandler method), 1002
do_postresize() (FrameWork.ScrolledWindow method), 1413
do_postresize() (FrameWork.Window method), 1412
do_update() (FrameWork.Window method), 1412
doc_header (cmd.Cmd attribute), 1069
DocCGIXMLRPCRequestHandler (class in DocXMLRPCServer), 1027
DocFileSuite() (in module doctest), 1166
doCleanups() (unittest.TestCase method), 1193
docmd() (smtplib.SMTP method), 974
docstring, 1453
docstring (doctest.DocTest attribute), 1169
DocTest (class in doctest), 1168
doctest (module), 1152
DocTestFailure, 1175
DocTestFinder (class in doctest), 1170
DocTestParser (class in doctest), 1171
DocTestRunner (class in doctest), 1171
DocTestSuite() (in module doctest), 1167
doctype() (xml.etree.ElementTree.TreeBuilder method), 871
doctype() (xml.etree.ElementTree.XMLParser method), 871
documentation
    generation, 1151
    online, 1151
documentElement (xml.dom.Document attribute), 877
DocXMLRPCRequestHandler (class in DocXMLRPCServer), 1027
DocXMLRPCServer (class in DocXMLRPCServer), 1027
DocXMLRPCServer (module), 1027
domain_initial_dot (cookielib.Cookie attribute), 1010
domain_return_ok() (cookielib.CookiePolicy method), 1006
domain_specified (cookielib.Cookie attribute), 1010
DomainLiberal (cookielib.DefaultCookiePolicy attribute), 1009
DomainRFC2965Match (cookielib.DefaultCookiePolicy attribute), 1009
DomainStrict (cookielib.DefaultCookiePolicy attribute), 1009
DomainStrictNoDots (cookielib.DefaultCookiePolicy attribute), 1009
DomainStrictNonDomain (cookielib.DefaultCookiePolicy attribute), 1009
DOMEventStream (class in xml.dom.pulldom), 888
DOMException, 880
DomstringSizeErr, 881
done() (in module turtle), 1128
done() (xdrlib.Unpacker method), 405
DONT_ACCEPT_BLANKLINE (in module doctest), 1160
DONT_ACCEPT_TRUE_FOR_1 (in module doctest), 1160

dont_write_bytecode (in module sys), 1248
doRollover() (logging.handlers.RotatingFileHandler method), 547
doRollover() (logging.handlers.TimedRotatingFileHandler method), 548
DOT (in module token), 1333
dot() (in module turtle), 1116
DOTALL (in module re), 94
doublequote (csv.Dialect attribute), 391
DOUBLESLASH (in module token), 1333
DOUBLESLASHEQUAL (in module token), 1333
DOUBLESTAR (in module token), 1333
DOUBLESTAREQUAL (in module token), 1333
doupdate() (in module curses), 555
down() (in module turtle), 1120
drop_whitespace (textwrap.TextWrapper attribute), 123
dropwhile() (in module itertools), 270
dst() (datetime.datetime method), 156
dst() (datetime.time method), 161
dst() (datetime.tzinfo method), 162
DTDHandler (class in xml.sax.handler), 890
duck-typing, 1453
dumbdbm
    module, 333
dumbdbm (module), 341
DumbWriter (class in formatter), 1362
dummy_thread (module), 639
dummy_threading (module), 638
dump() (in module ast), 1330
dump() (in module json), 799
dump() (in module marshal), 332
dump() (in module pickle), 319
dump() (in module xml.etree.ElementTree), 865
dump() (pickle.Pickler method), 320
dump_address_pair() (in module rfc822), 837
dump_stats() (profile.Profile method), 1228
dump_stats() (pstats.Stats method), 1229
dumps() (in module json), 800
dumps() (in module marshal), 332
dumps() (in module pickle), 319
dumps() (in module xmlrpclib), 1022
dup() (in module os), 422
dup() (posixfile.posixfile method), 1395
dup2() (in module os), 422
dup2() (posixfile.posixfile method), 1395
DUP_TOP (opcode), 1343
DUP_TOPX (opcode), 1347
DuplicateSectionError, 397
dwFlags (subprocess.STARTUPINFO attribute), 708
DynLoadSuffixImporter (class in imputil), 1310

E

e (in module cmath), 233
e (in module math), 230
E2BIG (in module errno), 580
EACCES (in module errno), 580
EADDRINUSE (in module errno), 585
EADDRNOTAVAIL (in module errno), 585
EADV (in module errno), 583
EAFNOSUPPORT (in module errno), 584
EAFP, 1453
EAGAIN (in module errno), 580
EALREADY (in module errno), 585
east_asian_width() (in module unicodedata), 140
EasyDialogs (module), 1407
EBADE (in module errno), 582
EBADF (in module errno), 580
EBADFD (in module errno), 584
EBADMSG (in module errno), 583
EBADR (in module errno), 582
EBADRQC (in module errno), 583
EBADSLT (in module errno), 583
EBFONT (in module errno), 583
EBUSY (in module errno), 581
ECHILD (in module errno), 580
echo() (in module curses), 555
echochar() (curses.window method), 563
ECHRNG (in module errno), 582
ECOMM (in module errno), 583
ECONNABORTED (in module errno), 585
ECONNREFUSED (in module errno), 585
ECONNRESET (in module errno), 585
EDEADLK (in module errno), 582
EDEADLOCK (in module errno), 583
EDESTADDRREQ (in module errno), 584
edit() (curses.textpad.Textbox method), 572
EDOM (in module errno), 581
EDOTDOT (in module errno), 583
EDQUOT (in module errno), 586
EEXIST (in module errno), 581
EFAULT (in module errno), 581
EFBIG (in module errno), 581
effective() (in module bdb), 1219
ehlo() (smtplib.SMTP method), 974
ehlo_or_helo_if_needed() (smtplib.SMTP method), 975
EHOSTDOWN (in module errno), 585
EHOSTUNREACH (in module errno), 585
EIDRM (in module errno), 582
EILSEQ (in module errno), 584
EINPROGRESS (in module errno), 585
EINTR (in module errno), 580
EINVAL (in module errno), 581
EIO (in module errno), 580
EISCONN (in module errno), 585
EISDIR (in module errno), 581
EISNAM (in module errno), 586

EL2HLT (in module errno), 582
EL2NSYNC (in module errno), 582
EL3HLT (in module errno), 582
EL3RST (in module errno), 582
Element (class in xml.etree.ElementTree), 867
element_create() (ttk.Style method), 1102
element_names() (ttk.Style method), 1103
element_options() (ttk.Style method), 1103
ElementDeclHandler() (xml.parsers.expat.xmlparser method), 904
elements() (collections.Counter method), 175
ElementTree (class in xml.etree.ElementTree), 869
ELIBACC (in module errno), 584
ELIBBAD (in module errno), 584
ELIBEXEC (in module errno), 584
ELIBMAX (in module errno), 584
ELIBSCN (in module errno), 584
Ellinghouse, Lance, 845
Ellipsis (built-in variable), 29
ELLIPSIS (in module doctest), 1160
EllipsisType (in module types), 215
ELNRNG (in module errno), 582
ELOOP (in module errno), 582
email (module), 765
email.charset (module), 781
email.encoders (module), 784
email.errors (module), 784
email.generator (module), 775
email.header (module), 779
email.iterators (module), 787
email.message (module), 765
email.mime (module), 777
email.parser (module), 772
email.utils (module), 785
EMFILE (in module errno), 581
emit() (logging.FileHandler method), 546
emit() (logging.Handler method), 526
emit() (logging.handlers.BufferingHandler method), 552
emit() (logging.handlers.DatagramHandler method), 549
emit() (logging.handlers.HTTPHandler method), 553
emit() (logging.handlers.NTEventLogHandler method), 551
emit() (logging.handlers.RotatingFileHandler method), 547
emit() (logging.handlers.SMTPHandler method), 552
emit() (logging.handlers.SocketHandler method), 548
emit() (logging.handlers.SysLogHandler method), 550
emit() (logging.handlers.TimedRotatingFileHandler method), 548
emit() (logging.handlers.WatchedFileHandler method), 546
emit() (logging.NullHandler method), 546
emit() (logging.StreamHandler method), 545
EMLINK (in module errno), 581
Empty, 204
empty() (multiprocessing.multiprocessing.queues.SimpleQueue method), 649
empty() (multiprocessing.Queue method), 648
empty() (Queue.Queue method), 205
empty() (sched.scheduler method), 203
EMPTY_NAMESPACE (in module xml.dom), 873
emptyline() (cmd.Cmd method), 1068
EMSGSIZE (in module errno), 584
EMULTIHOP (in module errno), 583
enable() (bdb.Breakpoint method), 1215
enable() (in module cgitb), 921
enable() (in module gc), 1281
enable() (profile.Profile method), 1228
enable_callback_tracebacks() (in module sqlite3), 345
enable_interspersed_args() (optparse.OptionParser method), 510
enable_load_extension() (sqlite3.Connection method), 348
enable_traversal() (ttk.Notebook method), 1093
ENABLE_USER_SITE (in module site), 1291
EnableReflectionKey() (in module _winreg), 1377
ENAMETOOLONG (in module errno), 582
ENAVAIL (in module errno), 586
enclose() (curses.window method), 563
encode
    Codecs, 124
encode() (codecs.Codec method), 129
encode() (codecs.IncrementalEncoder method), 130
encode() (email.header.Header method), 780
encode() (in module base64), 841
encode() (in module codecs), 124
encode() (in module mimetools), 828
encode() (in module quopri), 845
encode() (in module uu), 845
encode() (json.JSONEncoder method), 803
encode() (str method), 40
encode() (xmlrpclib.Binary method), 1019
encode() (xmlrpclib.Boolean method), 1017
encode() (xmlrpclib.DateTime method), 1018
encode_7or8bit() (in module email.encoders), 784
encode_base64() (in module email.encoders), 784
encode_noop() (in module email.encoders), 784
encode_quopri() (in module email.encoders), 784
encode_rfc2231() (in module email.utils), 786
encode_threshold (SimpleXMLRPCServer.SimpleXMLRPCRequestH attribute), 1025
encoded_header_len() (email.charset.Charset method), 782
EncodedFile() (in module codecs), 127

encodePriority() (logging.handlers.SysLogHandler method), 550
encodestring() (in module base64), 841
encodestring() (in module quopri), 845
encoding
  base64, 840
  quoted-printable, 844
encoding (exceptions.UnicodeError attribute), 71
encoding (file attribute), 60
ENCODING (in module tarfile), 378
encoding (io.TextIOBase attribute), 453
encodings.idna (module), 138
encodings.utf_8_sig (module), 139
encodings_map (in module mimetypes), 830
encodings_map (mimetypes.MimeTypes attribute), 831
end (exceptions.UnicodeError attribute), 71
end() (re.MatchObject method), 100
end() (xml.etree.ElementTree.TreeBuilder method), 871
end_fill() (in module turtle), 1123
END_FINALLY (opcode), 1346
end_group() (fl.form method), 1435
end_headers() (BaseHTTPServer.BaseHTTPRequestHandler method), 998
end_marker() (multifile.MultiFile method), 835
end_paragraph() (formatter.formatter method), 1359
end_poly() (in module turtle), 1128
EndCdataSectionHandler() (xml.parsers.expat.xmlparser method), 905
EndDoctypeDeclHandler() (xml.parsers.expat.xmlparser method), 904
endDocument() (xml.sax.handler.ContentHandler method), 892
endElement() (xml.sax.handler.ContentHandler method), 893
EndElementHandler() (xml.parsers.expat.xmlparser method), 904
endElementNS() (xml.sax.handler.ContentHandler method), 893
endheaders() (httplib.HTTPConnection method), 954
ENDMARKER (in module token), 1333
EndNamespaceDeclHandler() (xml.parsers.expat.xmlparser method), 905
endpick() (in module gl), 1440
endpos (re.MatchObject attribute), 100
endPrefixMapping() (xml.sax.handler.ContentHandler method), 893
endselect() (in module gl), 1440
endswith() (str method), 40
endwin() (in module curses), 556
ENETDOWN (in module errno), 585
ENETRESET (in module errno), 585
ENETUNREACH (in module errno), 585
ENFILE (in module errno), 581
ENOANO (in module errno), 582
ENOBUFS (in module errno), 585
ENOCSI (in module errno), 582
ENODATA (in module errno), 583
ENODEV (in module errno), 581
ENOENT (in module errno), 580
ENOEXEC (in module errno), 580
ENOLCK (in module errno), 582
ENOLINK (in module errno), 583
ENOMEM (in module errno), 580
ENOMSG (in module errno), 582
ENONET (in module errno), 583
ENOPKG (in module errno), 583
ENOPROTOOPT (in module errno), 584
ENOSPC (in module errno), 581
ENOSR (in module errno), 583
ENOSTR (in module errno), 583
ENOSYS (in module errno), 582
ENOTBLK (in module errno), 581
ENOTCONN (in module errno), 585
ENOTDIR (in module errno), 581
ENOTEMPTY (in module errno), 582
ENOTNAM (in module errno), 585
ENOTSOCK (in module errno), 584
ENOTTY (in module errno), 581
ENOTUNIQ (in module errno), 583
ensurepip (module), 1243
enter() (sched.scheduler method), 203
enterabs() (sched.scheduler method), 202
entities (xml.dom.DocumentType attribute), 877
EntityDeclHandler() (xml.parsers.expat.xmlparser method), 904
entitydefs (in module htmlentitydefs), 857
EntityResolver (class in xml.sax.handler), 890
Enum (class in aetypes), 1423
enum_certificates() (in module ssl), 732
enum_crls() (in module ssl), 732
enumerate() (built-in function), 9
enumerate() (in module fm), 1438
enumerate() (in module threading), 627
EnumKey() (in module _winreg), 1374
enumsubst() (in module aetools), 1421
EnumValue() (in module _winreg), 1374
environ (in module os), 416
environ (in module posix), 1384
environment variable
  <protocol>_proxy, 940
  AUDIODEV, 1044

BROWSER, 911, 912
COLUMNS, 560
COMSPEC, 441, 705
ftp_proxy, 931
HOME, 292, 1292
HOMEDRIVE, 292
HOMEPATH, 292
http_proxy, 931, 938, 948
IDLESTARTUP, 1146
KDEDIR, 913
LANG, 1049, 1051, 1059, 1062
LANGUAGE, 1049, 1051
LC_ALL, 1049, 1051
LC_MESSAGES, 1049, 1051
LINES, 556, 560
LNAME, 554
LOGNAME, 417, 554
MIXERDEV, 1044
no_proxy, 932
PAGER, 1151, 1222
PATH, 436, 439, 444, 911, 919, 921
POSIXLY_CORRECT, 520
PYTHON_DOM, 873
PYTHONDOCS, 1152
PYTHONDONTWRITEBYTECODE, 1248
PYTHONHASHSEED, 263
PYTHONNOUSERSITE, 1291
PYTHONPATH, 919, 1255
PYTHONSTARTUP, 698, 700, 1146, 1292
PYTHONUSERBASE, 1291
PYTHONY2K, 456, 457
SSL_CERT_FILE, 750
SSL_CERT_PATH, 750
SystemRoot, 706
TEMP, 306
TERM, 559
TIX_LIBRARY, 1105
TMP, 306, 433
TMPDIR, 306, 433
TZ, 461, 462
USER, 554
USERNAME, 554
USERPROFILE, 292
Wimp$ScrapDir, 306
environment variables
deleting, 420
setting, 418
EnvironmentError, 68
EnvironmentVarGuard (class in test.support), 1213
ENXIO (in module errno), 580
eof (shlex.shlex attribute), 1072
EOFError, 68
EOPNOTSUPP (in module errno), 584
EOVERFLOW (in module errno), 583
EPERM (in module errno), 580
EPFNOSUPPORT (in module errno), 584
epilogue (email.message.Message attribute), 772
EPIPE (in module errno), 581
epoch, 456
epoll() (in module select), 621
EPROTO (in module errno), 583
EPROTONOSUPPORT (in module errno), 584
EPROTOTYPE (in module errno), 584
eq() (in module operator), 282
EQEQUAL (in module token), 1333
EQUAL (in module token), 1333
ERA (in module locale), 1062
ERA_D_FMT (in module locale), 1062
ERA_D_T_FMT (in module locale), 1062
ERA_T_FMT (in module locale), 1062
ERANGE (in module errno), 581
erase() (curses.window method), 563
erasechar() (in module curses), 556
EREMCHG (in module errno), 584
EREMOTE (in module errno), 583
EREMOTEIO (in module errno), 586
ERESTART (in module errno), 584
erf() (in module math), 230
erfc() (in module math), 230
EROFS (in module errno), 581
ERR (in module curses), 567
errcheck (ctypes._FuncPtr attribute), 609
errcode (xmlrpclib.ProtocolError attribute), 1020
errmsg (xmlrpclib.ProtocolError attribute), 1020
errno
module, 69, 714
errno (module), 580
Error, 311, 391, 397, 406, 822, 842, 844, 846, 911, 1036, 1039, 1059, 1405
error, 97, 105, 217, 333, 335–337, 341, 363, 415, 520, 554, 621, 637, 714, 900, 1029, 1032, 1387, 1396, 1399, 1403, 1430, 1441, 1442, 1445
ERROR (in module cd), 1430
error() (argparse.ArgumentParser method), 491
error() (in module logging), 532
error() (logging.Logger method), 524
error() (mhlib.Folder method), 827
error() (mhlib.MH method), 826
error() (urllib2.OpenerDirector method), 943
error() (xml.sax.handler.ErrorHandler method), 895
error_body (wsgiref.handlers.BaseHandler attribute), 929
error_content_type (BaseHTTPServer.BaseHTTPRequestHandler attribute), 998
error_headers (wsgiref.handlers.BaseHandler attribute), 929
error_leader() (shlex.shlex method), 1071
error_message_format

(BaseHTTPServer.BaseHTTPRequestHandler attribute), 997
error_output() (wsgiref.handlers.BaseHandler method), 929
error_perm, 957
error_proto, 957, 961
error_reply, 957
error_status (wsgiref.handlers.BaseHandler attribute), 929
error_temp, 957
ErrorByteIndex (xml.parsers.expat.xmlparser attribute), 903
errorcode (in module errno), 580
ErrorCode (xml.parsers.expat.xmlparser attribute), 903
ErrorColumnNumber (xml.parsers.expat.xmlparser attribute), 903
ErrorHandler (class in xml.sax.handler), 890
ErrorLineNumber (xml.parsers.expat.xmlparser attribute), 903
Errors
  logging, 521
errors (file attribute), 60
errors (io.TextIOBase attribute), 453
errors (unittest.TestResult attribute), 1196
ErrorString() (in module xml.parsers.expat), 900
ERRORTOKEN (in module token), 1333
escape (shlex.shlex attribute), 1071
escape() (in module cgi), 918
escape() (in module re), 96
escape() (in module xml.sax.saxutils), 895
escapechar (csv.Dialect attribute), 391
escapedquotes (shlex.shlex attribute), 1071
ESHUTDOWN (in module errno), 585
ESOCKTNOSUPPORT (in module errno), 584
ESPIPE (in module errno), 581
ESRCH (in module errno), 580
ESRMNT (in module errno), 583
ESTALE (in module errno), 585
ESTRPIPE (in module errno), 584
ETIME (in module errno), 583
ETIMEDOUT (in module errno), 585
Etiny() (decimal.Context method), 248
ETOOMANYREFS (in module errno), 585
Etop() (decimal.Context method), 248
ETXTBSY (in module errno), 581
EUCLEAN (in module errno), 585
EUNATCH (in module errno), 582
EUSERS (in module errno), 584
eval
  built-in function, 65, 86, 219, 1323
eval() (built-in function), 9
Event (class in multiprocessing), 652
Event (class in threading), 635
event scheduling, 201
event() (msilib.Control method), 1369
Event() (multiprocessing.managers.SyncManager method), 659
events (widgets), 1084
EWOULDBLOCK (in module errno), 582
EX_CANTCREAT (in module os), 437
EX_CONFIG (in module os), 438
EX_DATAERR (in module os), 436
EX_IOERR (in module os), 437
EX_NOHOST (in module os), 437
EX_NOINPUT (in module os), 436
EX_NOPERM (in module os), 438
EX_NOTFOUND (in module os), 438
EX_NOUSER (in module os), 436
EX_OK (in module os), 436
EX_OSERR (in module os), 437
EX_OSFILE (in module os), 437
EX_PROTOCOL (in module os), 437
EX_SOFTWARE (in module os), 437
EX_TEMPFAIL (in module os), 437
EX_UNAVAILABLE (in module os), 437
EX_USAGE (in module os), 436
Example (class in doctest), 1169
example (doctest.DocTestFailure attribute), 1176
example (doctest.UnexpectedException attribute), 1176
examples (doctest.DocTest attribute), 1169
exc_clear() (in module sys), 1249
exc_info (doctest.UnexpectedException attribute), 1176
exc_info() (in module sys), 1248
exc_msg (doctest.Example attribute), 1169
exc_traceback (in module sys), 1249
exc_type (in module sys), 1249
exc_value (in module sys), 1249
excel (class in csv), 390
excel_tab (class in csv), 390
except
  statement, 67
except (2to3 fixer), 1205
excepthook() (in module sys), 921, 1248
Exception, 67
EXCEPTION (in module Tkinter), 1086
exception() (in module logging), 532
exception() (logging.Logger method), 524
exceptions
  in CGI scripts, 921
exceptions (module), 67
EXDEV (in module errno), 581
exec
  statement, 65
exec (2to3 fixer), 1205
exec_prefix (in module sys), 1249

EXEC_STMT (opcode), 1346
execfile
    built-in function, 1292
execfile (2to3 fixer), 1205
execfile() (built-in function), 10
execl() (in module os), 435
execle() (in module os), 435
execlp() (in module os), 435
execlpe() (in module os), 435
executable (in module sys), 1249
Execute() (msilib.View method), 1366
execute() (sqlite3.Connection method), 346
execute() (sqlite3.Cursor method), 351
executemany() (sqlite3.Connection method), 346
executemany() (sqlite3.Cursor method), 351
executescript() (sqlite3.Connection method), 346
executescript() (sqlite3.Cursor method), 352
execv() (in module os), 435
execve() (in module os), 435
execvp() (in module os), 435
execvpe() (in module os), 435
ExFileSelectBox (class in Tix), 1106
EXFULL (in module errno), 582
exists() (in module os.path), 292
exists() (ttk.Treeview method), 1098
exit (built-in variable), 29
exit() (argparse.ArgumentParser method), 491
exit() (in module sys), 1249
exit() (in module thread), 637
exitcode (multiprocessing.Process attribute), 646
exitfunc (2to3 fixer), 1205
exitfunc (in module sys), 1250
exitfunc (in sys), 1274
exitonclick() (in module turtle), 1135
exp() (decimal.Context method), 249
exp() (decimal.Decimal method), 240
exp() (in module cmath), 231
exp() (in module math), 228
expand() (re.MatchObject method), 98
expand_tabs (textwrap.TextWrapper attribute), 123
ExpandEnvironmentStrings() (in module _winreg), 1374
expandNode() (xml.dom.pulldom.DOMEventStream method), 888
expandtabs() (in module string), 86
expandtabs() (str method), 40
expanduser() (in module os.path), 292
expandvars() (in module os.path), 292
Expat, 900
ExpatError, 900
expect() (telnetlib.Telnet method), 980
expectedFailure() (in module unittest), 1186
expectedFailures (unittest.TestResult attribute), 1197
expires (cookielib.Cookie attribute), 1009
expm1() (in module math), 228
expovariate() (in module random), 264
expr() (in module parser), 1322
expression, 1453
expunge() (imaplib.IMAP4 method), 965
extend() (array.array method), 197
extend() (collections.deque method), 177
extend() (list method), 48
extend() (xml.etree.ElementTree.Element method), 868
extend_path() (in module pkgutil), 1314
extended slice
    assignment, 48
    operation, 38
EXTENDED_ARG (opcode), 1349
ExtendedContext (class in decimal), 246
extendleft() (collections.deque method), 177
extension module, 1453
extensions_map (SimpleHTTPServer.SimpleHTTPRequestHandler attribute), 1000
External Data Representation, 318, 404
external_attr (zipfile.ZipInfo attribute), 376
ExternalClashError, 822
ExternalEntityParserCreate() (xml.parsers.expat.xmlparser method), 902
ExternalEntityRefHandler() (xml.parsers.expat.xmlparser method), 905
extra (zipfile.ZipInfo attribute), 375
extract() (tarfile.TarFile method), 380
extract() (zipfile.ZipFile method), 372
extract_cookies() (cookielib.CookieJar method), 1004
extract_stack() (in module traceback), 1276
extract_tb() (in module traceback), 1276
extract_version (zipfile.ZipInfo attribute), 375
extractall() (tarfile.TarFile method), 380
extractall() (zipfile.ZipFile method), 373
ExtractError, 378
extractfile() (tarfile.TarFile method), 381
extsep (in module os), 444

F

F_BAVAIL (in module statvfs), 301
F_BFREE (in module statvfs), 301
F_BLOCKS (in module statvfs), 301
F_BSIZE (in module statvfs), 301
F_FAVAIL (in module statvfs), 301
F_FFREE (in module statvfs), 301
F_FILES (in module statvfs), 301
F_FLAG (in module statvfs), 302
F_FRSIZE (in module statvfs), 301

F_NAMEMAX (in module statvfs), 302
F_OK (in module os), 426
fabs() (in module math), 226
factorial() (in module math), 226
fail() (unittest.TestCase method), 1192
failfast (unittest.TestResult attribute), 1197
failureException (unittest.TestCase attribute), 1192
failures (unittest.TestResult attribute), 1197
False, 31, 65
false, 31
False (Built-in object), 31
False (built-in variable), 29
family (socket.socket attribute), 723
FancyURLopener (class in urllib), 935
fatalError() (xml.sax.handler.ErrorHandler method), 895
Fault (class in xmlrpclib), 1020
faultCode (xmlrpclib.Fault attribute), 1020
faultString (xmlrpclib.Fault attribute), 1020
fchdir() (in module os), 426
fchmod() (in module os), 422
fchown() (in module os), 422
FCICreate() (in module msilib), 1365
fcntl
    module, 58
fcntl (module), 1390
fcntl() (in module fcntl), 1390, 1394
fd() (in module turtle), 1113
fdatasync() (in module os), 422
fdopen() (in module os), 420
Feature (class in msilib), 1369
feature_external_ges (in module xml.sax.handler), 891
feature_external_pes (in module xml.sax.handler), 891
feature_namespace_prefixes (in module xml.sax.handler), 891
feature_namespaces (in module xml.sax.handler), 890
feature_string_interning (in module xml.sax.handler), 891
feature_validation (in module xml.sax.handler), 891
feed() (email.parser.FeedParser method), 773
feed() (HTMLParser.HTMLParser method), 848
feed() (sgmllib.SGMLParser method), 853
feed() (xml.etree.ElementTree.XMLParser method), 871
feed() (xml.sax.xmlreader.IncrementalParser method), 898
FeedParser (class in email.parser), 773
fetch() (imaplib.IMAP4 method), 965
Fetch() (msilib.View method), 1367
fetchall() (sqlite3.Cursor method), 353
fetchmany() (sqlite3.Cursor method), 352
fetchone() (sqlite3.Cursor method), 352
fflags (select.kevent attribute), 625
field_size_limit() (in module csv), 389
fieldnames (csv.csvreader attribute), 392
fields (uuid.UUID attribute), 981
fifo (class in asynchat), 762
file
    .ini, 395
    .pdbrc, 1222
    .pythonrc.py, 1292
    built-in function, 57
    byte-code, 1305, 1307, 1339
    configuration, 395
    copying, 309
    debugger configuration, 1222
    descriptor, 58
    large files, 1383
    mime.types, 830
    object, 57
    path configuration, 1290
    plist, 407
    temporary, 304
    user configuration, 1292
file (pyclbr.Class attribute), 1338
file (pyclbr.Function attribute), 1338
file ...
    compileall command line option, 1339
file control
    UNIX, 1390
file name
    temporary, 304
file object, 1453
file object
    POSIX, 1394
file() (built-in function), 10
file() (posixfile.posixfile method), 1395
file-like object, 1453
file_dispatcher (class in asyncore), 759
file_open() (urllib2.FileHandler method), 947
file_size (zipfile.ZipInfo attribute), 376
file_wrapper (class in asyncore), 759
filecmp (module), 302
fileConfig() (in module logging.config), 536
FileCookieJar (class in cookielib), 1003
FileEntry (class in Tix), 1106
FileHandler (class in logging), 545
FileHandler (class in urllib2), 941
FileInput (class in fileinput), 296
fileinput (module), 295
FileIO (class in io), 451
filelineno() (in module fileinput), 296
filename (cookielib.FileCookieJar attribute), 1005
filename (doctest.DocTest attribute), 1169
filename (zipfile.ZipInfo attribute), 375

filename() (in module fileinput), 296
filename_only (in module tabnanny), 1337
filenames
    pathname expansion, 307
    wildcard expansion, 308
fileno() (Connection method), 650
fileno() (file method), 58
fileno() (hotshot.Profile method), 1233
fileno() (httplib.HTTPResponse method), 954
fileno() (in module fileinput), 296
fileno() (io.IOBase method), 448
fileno() (ossaudiodev.oss_audio_device method), 1045
fileno() (ossaudiodev.oss_mixer_device method), 1047
fileno() (select.epoll method), 623
fileno() (select.kqueue method), 624
fileno() (socket.socket method), 720
fileno() (SocketServer.BaseServer method), 990
fileno() (telnetlib.Telnet method), 980
fileopen() (in module posixfile), 1394
FileSelectBox (class in Tix), 1106
FileType (class in argparse), 488
FileType (in module types), 215
FileWrapper (class in wsgiref.util), 923
fill() (in module textwrap), 122
fill() (in module turtle), 1123
fill() (textwrap.TextWrapper method), 124
fillcolor() (in module turtle), 1122
filter (2to3 fixer), 1205
Filter (class in logging), 527
filter (select.kevent attribute), 625
filter() (built-in function), 11
filter() (in module curses), 556
filter() (in module fnmatch), 308
filter() (in module future_builtins), 1264
filter() (logging.Filter method), 527
filter() (logging.Handler method), 526
filter() (logging.Logger method), 524
filterwarnings() (in module warnings), 1269
find() (doctest.DocTestFinder method), 1170
find() (in module gettext), 1051
find() (in module string), 86
find() (mmap.mmap method), 694
find() (str method), 40
find() (xml.etree.ElementTree.Element method), 868
find() (xml.etree.ElementTree.ElementTree method), 869
find_first() (fl.form method), 1435
find_global() (pickle protocol), 325
find_last() (fl.form method), 1435
find_library() (in module ctypes.util), 612
find_loader() (in module pkgutil), 1315
find_longest_match() (difflib.SequenceMatcher method), 114
find_module() (imp.NullImporter method), 1308
find_module() (in module imp), 1305
find_module() (zipimport.zipimporter method), 1313
find_msvcrt() (in module ctypes.util), 612
find_user_password() (urllib2.HTTPPasswordMgr method), 946
findall() (in module re), 95
findall() (re.RegexObject method), 98
findall() (xml.etree.ElementTree.Element method), 868
findall() (xml.etree.ElementTree.ElementTree method), 869
findCaller() (logging.Logger method), 525
finder, 1453
findertools (module), 1407
findfactor() (in module audioop), 1030
findfile() (in module test.support), 1211
findfit() (in module audioop), 1030
findfont() (in module fm), 1438
finditer() (in module re), 96
finditer() (re.RegexObject method), 98
findlabels() (in module dis), 1342
findlinestarts() (in module dis), 1342
findmatch() (in module mailcap), 805
findmax() (in module audioop), 1030
findtext() (xml.etree.ElementTree.Element method), 868
findtext() (xml.etree.ElementTree.ElementTree method), 869
finish() (SocketServer.BaseRequestHandler method), 992
finish_request() (SocketServer.BaseServer method), 991
first() (asynchat.fifo method), 762
first() (bsddb.bsddbobject method), 340
first() (dbhash.dbhash method), 338
firstChild (xml.dom.Node attribute), 875
firstkey() (in module gdbm), 336
firstweekday() (in module calendar), 172
fix() (in module fpformat), 143
fix_missing_locations() (in module ast), 1329
fix_sentence_endings (textwrap.TextWrapper attribute), 123
FL (module), 1437
fl (module), 1433
flag_bits (zipfile.ZipInfo attribute), 375
flags (in module sys), 1250
flags (re.RegexObject attribute), 98
flags (select.kevent attribute), 625
flags() (posixfile.posixfile method), 1395
flash() (in module curses), 556

flatten() (email.generator.Generator method), 775
flattening
    objects, 317
float
    built-in function, 33, 85
float (built-in class), 11
float_info (in module sys), 1250
float_repr_style (in module sys), 1251
floating point
    literals, 33
    object, 33
FloatingPointError, 68, 1293
FloatType (in module types), 214
flock() (in module fcntl), 1391
floor division, 1454
floor() (in module math), 34, 226
floordiv() (in module operator), 283
flp (module), 1438
flush() (bz2.BZ2Compressor method), 370
flush() (file method), 58
flush() (formatter.writer method), 1361
flush() (io.BufferedWriter method), 452
flush() (io.IOBase method), 448
flush() (logging.Handler method), 526
flush() (logging.handlers.BufferingHandler method), 552
flush() (logging.handlers.MemoryHandler method), 552
flush() (logging.StreamHandler method), 545
flush() (mailbox.Mailbox method), 809
flush() (mailbox.Maildir method), 810
flush() (mailbox.MH method), 813
flush() (mmap.mmap method), 694
flush() (zlib.Compress method), 365
flush() (zlib.Decompress method), 366
flush_softspace() (formatter.formatter method), 1360
flushheaders() (MimeWriter.MimeWriter method), 832
flushinp() (in module curses), 556
FlushKey() (in module _winreg), 1374
fm (module), 1438
fma() (decimal.Context method), 249
fma() (decimal.Decimal method), 241
fmod() (in module math), 227
fnmatch (module), 308
fnmatch() (in module fnmatch), 308
fnmatchcase() (in module fnmatch), 308
focus() (ttk.Treeview method), 1098
Folder (class in mhlib), 826
Font Manager, IRIS, 1438
fontpath() (in module fm), 1438
FOR_ITER (opcode), 1348
forget() (in module test.support), 1210
forget() (ttk.Notebook method), 1093
fork() (in module os), 438
fork() (in module pty), 1390
ForkingMixIn (class in SocketServer), 989
ForkingTCPServer (class in SocketServer), 989
ForkingUDPServer (class in SocketServer), 989
forkpty() (in module os), 438
Form (class in Tix), 1108
format
    str, 11
format (memoryview attribute), 62
format (struct.Struct attribute), 109
format() (built-in function), 11
format() (in module locale), 1063
format() (logging.Formatter method), 527
format() (logging.Handler method), 526
format() (pprint.PrettyPrinter method), 219
format() (str method), 41
format() (string.Formatter method), 76
format_exc() (in module traceback), 1276
format_exception() (in module traceback), 1277
format_exception_only() (in module traceback), 1276
format_field() (string.Formatter method), 77
format_help() (argparse.ArgumentParser method), 490
format_list() (in module traceback), 1276
format_stack() (in module traceback), 1277
format_stack_entry() (bdb.Bdb method), 1218
format_string() (in module locale), 1063
format_tb() (in module traceback), 1277
format_usage() (argparse.ArgumentParser method), 490
formataddr() (in module email.utils), 785
formatargspec() (in module inspect), 1288
formatargvalues() (in module inspect), 1288
formatdate() (in module email.utils), 786
FormatError, 823
FormatError() (in module ctypes), 613
formatException() (logging.Formatter method), 527
formatmonth() (calendar.HTMLCalendar method), 172
formatmonth() (calendar.TextCalendar method), 171
formatter
    module, 855
Formatter (class in logging), 527
Formatter (class in string), 76
formatter (htmllib.HTMLParser attribute), 856
formatter (module), 1359
formatTime() (logging.Formatter method), 527
formatting, string (%), 46
formatwarning() (in module warnings), 1268
formatyear() (calendar.HTMLCalendar method), 172
formatyear() (calendar.TextCalendar method), 171

formatyearpage() (calendar.HTMLCalendar method), 172
FORMS Library, 1433
forward() (in module turtle), 1113
found_terminator() (asynchat.async_chat method), 761
fp (rfc822.Message attribute), 839
fpathconf() (in module os), 422
fpectl (module), 1293
fpformat (module), 142
Fraction (class in fractions), 260
fractions (module), 260
frame (ScrolledText.ScrolledText attribute), 1109
FrameType (in module types), 215
FrameWork
    module, 1424
FrameWork (module), 1410
freeze_form() (fl.form method), 1435
freeze_support() (in module multiprocessing), 650
frexp() (in module math), 227
from_address() (ctypes._CData method), 614
from_buffer() (ctypes._CData method), 614
from_buffer_copy() (ctypes._CData method), 614
from_decimal() (fractions.Fraction method), 261
from_float() (decimal.Decimal method), 240
from_float() (fractions.Fraction method), 261
from_iterable() (itertools.chain class method), 268
from_param() (ctypes._CData method), 614
from_splittable() (email.charset.Charset method), 782
frombuf() (tarfile.TarInfo method), 382
fromchild (popen2.Popen3 attribute), 755
fromfd() (in module socket), 718
fromfd() (select.epoll method), 623
fromfd() (select.kqueue method), 624
fromfile() (array.array method), 197
fromhex() (float method), 36
fromkeys() (collections.Counter method), 175
fromkeys() (dict method), 54
fromlist() (array.array method), 197
fromordinal() (datetime.date class method), 149
fromordinal() (datetime.datetime class method), 153
fromstring() (array.array method), 197
fromstring() (in module xml.etree.ElementTree), 865
fromstringlist() (in module xml.etree.ElementTree), 865
fromtarfile() (tarfile.TarInfo method), 382
fromtimestamp() (datetime.date class method), 149
fromtimestamp() (datetime.datetime class method), 153
fromunicode() (array.array method), 197
fromutc() (datetime.tzinfo method), 163
frozenset (built-in class), 50
fstat() (in module os), 423
fstatvfs() (in module os), 423
fsum() (in module math), 227
fsync() (in module os), 423
FTP, 936
    ftplib (standard module), 956
    protocol, 936, 956
FTP (class in ftplib), 956
ftp_open() (urllib2.FTPHandler method), 947
ftp_proxy, 931
FTP_TLS (class in ftplib), 956
FTPHandler (class in urllib2), 941
ftplib (module), 956
ftpmirror.py, 957
ftruncate() (in module os), 423
Full, 204
full() (multiprocessing.Queue method), 648
full() (Queue.Queue method), 205
func (functools.partial attribute), 282
func_code (function object attribute), 65
funcattrs (2to3 fixer), 1205
function, 1454
Function (class in symtable), 1331
function() (in module new), 216
FunctionTestCase (class in unittest), 1193
FunctionType (in module types), 214
functools (module), 279
funny_files (filecmp.dircmp attribute), 303
future (2to3 fixer), 1205
future_builtins (module), 1263
FutureWarning, 72

G

G.722, 1035
gaierror, 714
gamma() (in module math), 230
gammavariate() (in module random), 264
garbage (in module gc), 1282
garbage collection, 1454
gather() (curses.textpad.Textbox method), 572
gauss() (in module random), 265
gc (module), 1281
gcd() (in module fractions), 261
gdbm
    module, 330, 333
gdbm (module), 335
ge() (in module operator), 282
gen_uuid() (in module msilib), 1366
generate_tokens() (in module tokenize), 1335
generator, 1454
generator, 1454
Generator (class in email.generator), 775
generator expression, 1454
generator expression, 1454
GeneratorExit, 68

GeneratorType (in module types), 214
generic_visit() (ast.NodeVisitor method), 1329
genops() (in module pickletools), 1350
gensuitemodule (module), 1420
get() (ConfigParser.ConfigParser method), 399
get() (ConfigParser.RawConfigParser method), 398
get() (dict method), 54
get() (email.message.Message method), 768
get() (in module webbrowser), 912
get() (mailbox.Mailbox method), 808
get() (multiprocessing.multiprocessing.queues.SimpleQueue method), 649
get() (multiprocessing.pool.AsyncResult method), 664
get() (multiprocessing.Queue method), 648
get() (ossaudiodev.oss_mixer_device method), 1047
get() (Queue.Queue method), 205
get() (rfc822.Message method), 838
get() (ttk.Combobox method), 1091
get() (xml.etree.ElementTree.Element method), 867
get_all() (email.message.Message method), 768
get_all() (wsgiref.headers.Headers method), 924
get_all_breaks() (bdb.Bdb method), 1218
get_app() (wsgiref.simple_server.WSGIServer method), 925
get_archive_formats() (in module shutil), 313
get_begidx() (in module readline), 698
get_body_encoding() (email.charset.Charset method), 782
get_boundary() (email.message.Message method), 771
get_break() (bdb.Bdb method), 1218
get_breaks() (bdb.Bdb method), 1218
get_buffer() (xdrlib.Packer method), 404
get_buffer() (xdrlib.Unpacker method), 405
get_ca_certs() (ssl.SSLContext method), 740
get_channel_binding() (ssl.SSLSocket method), 738
get_charset() (email.message.Message method), 767
get_charsets() (email.message.Message method), 771
get_children() (symtable.SymbolTable method), 1331
get_children() (ttk.Treeview method), 1097
get_close_matches() (in module difflib), 112
get_code() (imputil.BuiltinImporter method), 1310
get_code() (imputil.Importer method), 1309
get_code() (zipimport.zipimporter method), 1313
get_completer() (in module readline), 698
get_completer_delims() (in module readline), 698
get_completion_type() (in module readline), 698
get_config_h_filename() (in module sysconfig), 1262
get_config_var() (in module sysconfig), 1260
get_config_vars() (in module sysconfig), 1260
get_content_charset() (email.message.Message method), 771
get_content_maintype() (email.message.Message method), 769
get_content_subtype() (email.message.Message method), 769
get_content_type() (email.message.Message method), 769
get_count() (in module gc), 1281
get_current_history_length() (in module readline), 697
get_data() (in module pkgutil), 1316
get_data() (urllib2.Request method), 941
get_data() (zipimport.zipimporter method), 1313
get_date() (mailbox.MaildirMessage method), 816
get_debug() (in module gc), 1281
get_default() (argparse.ArgumentParser method), 490
get_default_domain() (in module nis), 1399
get_default_type() (email.message.Message method), 769
get_default_verify_paths() (in module ssl), 731
get_dialect() (in module csv), 388
get_directory() (in module fl), 1434
get_docstring() (in module ast), 1329
get_doctest() (doctest.DocTestParser method), 1171
get_endidx() (in module readline), 698
get_environ() (wsgiref.simple_server.WSGIRequestHandler method), 926
get_errno() (in module ctypes), 613
get_examples() (doctest.DocTestParser method), 1171
get_field() (string.Formatter method), 77
get_file() (mailbox.Babyl method), 813
get_file() (mailbox.Mailbox method), 808
get_file() (mailbox.Maildir method), 811
get_file() (mailbox.mbox method), 811
get_file() (mailbox.MH method), 813
get_file() (mailbox.MMDF method), 814
get_file_breaks() (bdb.Bdb method), 1218
get_filename() (email.message.Message method), 771
get_filename() (in module fl), 1434
get_filename() (zipimport.zipimporter method), 1313
get_flags() (mailbox.MaildirMessage method), 815
get_flags() (mailbox.mboxMessage method), 817
get_flags() (mailbox.MMDFMessage method), 821
get_folder() (mailbox.Maildir method), 810
get_folder() (mailbox.MH method), 812
get_frees() (symtable.Function method), 1331
get_from() (mailbox.mboxMessage method), 817
get_from() (mailbox.MMDFMessage method), 821
get_full_url() (urllib2.Request method), 942

get_globals() (symtable.Function method), 1331
get_grouped_opcodes() (difflib.SequenceMatcher method), 116
get_header() (urllib2.Request method), 942
get_history_item() (in module readline), 697
get_history_length() (in module readline), 696
get_host() (urllib2.Request method), 942
get_id() (symtable.SymbolTable method), 1331
get_ident() (in module thread), 637
get_identifiers() (symtable.SymbolTable method), 1331
get_importer() (in module pkgutil), 1315
get_info() (mailbox.MaildirMessage method), 816
GET_ITER (opcode), 1343
get_labels() (mailbox.Babyl method), 813
get_labels() (mailbox.BabylMessage method), 820
get_last_error() (in module ctypes), 613
get_line_buffer() (in module readline), 696
get_lineno() (symtable.SymbolTable method), 1331
get_loader() (in module pkgutil), 1315
get_locals() (symtable.Function method), 1331
get_logger() (in module multiprocessing), 668
get_magic() (in module imp), 1305
get_makefile_filename() (in module sysconfig), 1262
get_matching_blocks() (difflib.SequenceMatcher method), 115
get_message() (mailbox.Mailbox method), 808
get_method() (urllib2.Request method), 941
get_methods() (symtable.Class method), 1332
get_mouse() (in module fl), 1434
get_name() (symtable.Symbol method), 1332
get_name() (symtable.SymbolTable method), 1331
get_namespace() (symtable.Symbol method), 1332
get_namespaces() (symtable.Symbol method), 1332
get_nonstandard_attr() (cookielib.Cookie method), 1010
get_nowait() (multiprocessing.Queue method), 648
get_nowait() (Queue.Queue method), 205
get_objects() (in module gc), 1281
get_opcodes() (difflib.SequenceMatcher method), 115
get_option() (optparse.OptionParser method), 510
get_option_group() (optparse.OptionParser method), 501
get_origin_req_host() (urllib2.Request method), 942
get_osfhandle() (in module msvcrt), 1371
get_output_charset() (email.charset.Charset method), 782
get_param() (email.message.Message method), 770
get_parameters() (symtable.Function method), 1331
get_params() (email.message.Message method), 770
get_path() (in module sysconfig), 1261
get_path_names() (in module sysconfig), 1261
get_paths() (in module sysconfig), 1261
get_pattern() (in module fl), 1434
get_payload() (email.message.Message method), 766
get_platform() (in module sysconfig), 1261
get_poly() (in module turtle), 1128
get_position() (xdrlib.Unpacker method), 405
get_python_version() (in module sysconfig), 1261
get_recsrc() (ossaudiodev.oss_mixer_device method), 1048
get_referents() (in module gc), 1282
get_referrers() (in module gc), 1282
get_request() (SocketServer.BaseServer method), 991
get_rgbmode() (in module fl), 1433
get_scheme() (wsgiref.handlers.BaseHandler method), 929
get_scheme_names() (in module sysconfig), 1261
get_selector() (urllib2.Request method), 942
get_sequences() (mailbox.MH method), 812
get_sequences() (mailbox.MHMessage method), 819
get_server() (multiprocessing.managers.BaseManager method), 657
get_server_certificate() (in module ssl), 731
get_socket() (telnetlib.Telnet method), 980
get_source() (zipimport.zipimporter method), 1313
get_stack() (bdb.Bdb method), 1218
get_starttag_text() (HTMLParser.HTMLParser method), 849
get_starttag_text() (sgmllib.SGMLParser method), 853
get_stderr() (wsgiref.handlers.BaseHandler method), 928
get_stderr() (wsgiref.simple_server.WSGIRequestHandler method), 926
get_stdin() (wsgiref.handlers.BaseHandler method), 928
get_string() (mailbox.Mailbox method), 808
get_subdir() (mailbox.MaildirMessage method), 815
get_suffixes() (in module imp), 1305
get_symbols() (symtable.SymbolTable method), 1331
get_terminator() (asynchat.async_chat method), 761
get_threshold() (in module gc), 1282
get_token() (shlex.shlex method), 1070
get_type() (symtable.SymbolTable method), 1331
get_type() (urllib2.Request method), 942
get_unixfrom() (email.message.Message method), 766
get_usage() (optparse.OptionParser method), 512
get_value() (string.Formatter method), 77
get_version() (optparse.OptionParser method), 501
get_visible() (mailbox.BabylMessage method), 820

getabouttext() (FrameWork.Application method), 1411
getacl() (imaplib.IMAP4 method), 965
getaddr() (rfc822.Message method), 838
getaddresses() (in module email.utils), 785
getaddrinfo() (in module socket), 716
getaddrlist() (rfc822.Message method), 838
getallmatchingheaders() (rfc822.Message method), 838
getannotation() (imaplib.IMAP4 method), 965
getargspec() (in module inspect), 1287
GetArgv() (in module EasyDialogs), 1408
getargvalues() (in module inspect), 1287
getatime() (in module os.path), 292
getattr() (built-in function), 11
getAttribute() (xml.dom.Element method), 878
getAttributeNode() (xml.dom.Element method), 878
getAttributeNodeNS() (xml.dom.Element method), 878
getAttributeNS() (xml.dom.Element method), 878
GetBase() (xml.parsers.expat.xmlparser method), 902
getbegyx() (curses.window method), 563
getbkgd() (curses.window method), 563
getboolean() (ConfigParser.RawConfigParser method), 398
getByteStream() (xml.sax.xmlreader.InputSource method), 899
getcallargs() (in module inspect), 1288
getcanvas() (in module turtle), 1134
getcaps() (in module mailcap), 806
getch() (curses.window method), 563
getch() (in module msvcrt), 1371
getCharacterStream() (xml.sax.xmlreader.InputSource method), 899
getche() (in module msvcrt), 1371
getcheckinterval() (in module sys), 1251
getChild() (logging.Logger method), 523
getChildNodes() (compiler.ast.Node method), 1353
getChildren() (compiler.ast.Node method), 1353
getchildren() (xml.etree.ElementTree.Element method), 868
getclasstree() (in module inspect), 1287
GetColor() (in module ColorPicker), 1417
GetColumnInfo() (msilib.View method), 1366
getColumnNumber() (xml.sax.xmlreader.Locator method), 898
getcomments() (in module inspect), 1287
getcompname() (aifc.aifc method), 1034
getcompname() (sunau.AU_read method), 1037
getcompname() (wave.Wave_read method), 1039
getcomptype() (aifc.aifc method), 1034
getcomptype() (sunau.AU_read method), 1037
getcomptype() (wave.Wave_read method), 1039
getContentHandler() (xml.sax.xmlreader.XMLReader method), 897
getcontext() (in module decimal), 245
getcontext() (mhlib.MH method), 826
GetCreatorAndType() (in module MacOS), 1405
getctime() (in module os.path), 292
getcurrent() (mhlib.Folder method), 827
getcwd() (in module os), 426
getcwdu (2to3 fixer), 1205
getcwdu() (in module os), 426
getdate() (rfc822.Message method), 838
getdate_tz() (rfc822.Message method), 839
getdecoder() (in module codecs), 126
getdefaultencoding() (in module sys), 1251
getdefaultlocale() (in module locale), 1062
getdefaulttimeout() (in module socket), 719
getdlopenflags() (in module sys), 1252
getdoc() (in module inspect), 1287
getDOMImplementation() (in module xml.dom), 873
getDTDHandler() (xml.sax.xmlreader.XMLReader method), 897
getEffectiveLevel() (logging.Logger method), 523
getegid() (in module os), 416
getElementsByTagName() (xml.dom.Document method), 878
getElementsByTagName() (xml.dom.Element method), 878
getElementsByTagNameNS() (xml.dom.Document method), 878
getElementsByTagNameNS() (xml.dom.Element method), 878
getencoder() (in module codecs), 126
getencoding() (mimetools.Message method), 828
getEncoding() (xml.sax.xmlreader.InputSource method), 899
getEntityResolver() (xml.sax.xmlreader.XMLReader method), 897
getenv() (in module os), 418
getErrorHandler() (xml.sax.xmlreader.XMLReader method), 897
GetErrorString() (in module MacOS), 1405
geteuid() (in module os), 416
getEvent() (xml.dom.pulldom.DOMEventStream method), 888
getEventCategory() (logging.handlers.NTEventLogHandler method), 551
getEventType() (logging.handlers.NTEventLogHandler method), 551
getException() (xml.sax.SAXException method), 890
getFeature() (xml.sax.xmlreader.XMLReader method), 898
GetFieldCount() (msilib.Record method), 1367
getfile() (in module inspect), 1287

getfilesystemencoding() (in module sys), 1252
getfirst() (cgi.FieldStorage method), 917
getfirstmatchingheader() (rfc822.Message method), 838
getfloat() (ConfigParser.RawConfigParser method), 398
getfmts() (ossaudiodev.oss_audio_device method), 1045
getfqdn() (in module socket), 716
getframeinfo() (in module inspect), 1289
getframerate() (aifc.aifc method), 1034
getframerate() (sunau.AU_read method), 1037
getframerate() (wave.Wave_read method), 1039
getfullname() (mhlib.Folder method), 827
getgid() (in module os), 416
getgrall() (in module grp), 1386
getgrgid() (in module grp), 1386
getgrnam() (in module grp), 1386
getgroups() (in module os), 416
getheader() (httplib.HTTPResponse method), 954
getheader() (rfc822.Message method), 838
getheaders() (httplib.HTTPResponse method), 954
gethostbyaddr() (in module socket), 419, 717
gethostbyname() (in module socket), 716
gethostbyname_ex() (in module socket), 717
gethostname() (in module socket), 419, 717
getincrementaldecoder() (in module codecs), 126
getincrementalencoder() (in module codecs), 126
getinfo() (zipfile.ZipFile method), 372
getinnerframes() (in module inspect), 1289
GetInputContext() (xml.parsers.expat.xmlparser method), 902
getint() (ConfigParser.RawConfigParser method), 398
GetInteger() (msilib.Record method), 1367
getitem() (in module operator), 285
getiterator() (xml.etree.ElementTree.Element method), 868
getiterator() (xml.etree.ElementTree.ElementTree method), 869
getitimer() (in module signal), 753
getkey() (curses.window method), 563
getlast() (mhlib.Folder method), 827
GetLastError() (in module ctypes), 613
getLength() (xml.sax.xmlreader.Attributes method), 899
getLevelName() (in module logging), 533
getline() (in module linecache), 309
getLineNumber() (xml.sax.xmlreader.Locator method), 898
getlist() (cgi.FieldStorage method), 917
getloadavg() (in module os), 443
getlocale() (in module locale), 1062
getLogger() (in module logging), 531
getLoggerClass() (in module logging), 531
getlogin() (in module os), 417
getmaintype() (mimetools.Message method), 829
getmark() (aifc.aifc method), 1034
getmark() (sunau.AU_read method), 1037
getmark() (wave.Wave_read method), 1039
getmarkers() (aifc.aifc method), 1034
getmarkers() (sunau.AU_read method), 1037
getmarkers() (wave.Wave_read method), 1039
getmaxyx() (curses.window method), 563
getmcolor() (in module fl), 1434
getmember() (tarfile.TarFile method), 380
getmembers() (in module inspect), 1285
getmembers() (tarfile.TarFile method), 380
getMessage() (logging.LogRecord method), 528
getMessage() (xml.sax.SAXException method), 890
getmessagefilename() (mhlib.Folder method), 827
getMessageID() (logging.handlers.NTEventLogHandler method), 551
getmodule() (in module inspect), 1287
getmoduleinfo() (in module inspect), 1285
getmodulename() (in module inspect), 1285
getmouse() (in module curses), 556
getmro() (in module inspect), 1288
getmtime() (in module os.path), 292
getname() (chunk.Chunk method), 1041
getName() (threading.Thread method), 630
getNameByQName() (xml.sax.xmlreader.AttributesNS method), 900
getnameinfo() (in module socket), 717
getnames() (tarfile.TarFile method), 380
getNames() (xml.sax.xmlreader.Attributes method), 899
getnchannels() (aifc.aifc method), 1034
getnchannels() (sunau.AU_read method), 1037
getnchannels() (wave.Wave_read method), 1039
getnframes() (aifc.aifc method), 1034
getnframes() (sunau.AU_read method), 1037
getnframes() (wave.Wave_read method), 1039
getnode, 982
getnode() (in module uuid), 982
getopt (module), 519
getopt() (in module getopt), 519
GetoptError, 520
getouterframes() (in module inspect), 1289
getoutput() (in module commands), 1401
getpagesize() (in module resource), 1398
getparam() (mimetools.Message method), 828
getparams() (aifc.aifc method), 1034
getparams() (in module al), 1427
getparams() (sunau.AU_read method), 1037
getparams() (wave.Wave_read method), 1039
getparyx() (curses.window method), 563
getpass (module), 553

getpass() (in module getpass), 553
GetPassWarning, 553
getpath() (mhlib.MH method), 826
getpeercert() (ssl.SSLSocket method), 737
getpeername() (socket.socket method), 720
getpen() (in module turtle), 1128
getpgid() (in module os), 417
getpgrp() (in module os), 417
getpid() (in module os), 417
getplist() (mimetools.Message method), 828
getpos() (HTMLParser.HTMLParser method), 849
getppid() (in module os), 417
getpreferredencoding() (in module locale), 1062
getprofile() (in module sys), 1252
getprofile() (mhlib.MH method), 826
GetProperty() (msilib.SummaryInformation method), 1367
getProperty() (xml.sax.xmlreader.XMLReader method), 898
GetPropertyCount() (msilib.SummaryInformation method), 1367
getprotobyname() (in module socket), 717
getproxies() (in module urllib), 934
getPublicId() (xml.sax.xmlreader.InputSource method), 899
getPublicId() (xml.sax.xmlreader.Locator method), 898
getpwall() (in module pwd), 1384
getpwnam() (in module pwd), 1384
getpwuid() (in module pwd), 1384
getQNameByName() (xml.sax.xmlreader.AttributesNS method), 900
getQNames() (xml.sax.xmlreader.AttributesNS method), 900
getquota() (imaplib.IMAP4 method), 965
getquotaroot() (imaplib.IMAP4 method), 965
getrandbits() (in module random), 263
getrawheader() (rfc822.Message method), 838
getreader() (in module codecs), 126
getrecursionlimit() (in module sys), 1252
getrefcount() (in module sys), 1252
getresgid() (in module os), 417
getresponse() (httplib.HTTPConnection method), 953
getresuid() (in module os), 417
getrlimit() (in module resource), 1396
getroot() (xml.etree.ElementTree.ElementTree method), 869
getrusage() (in module resource), 1397
getsample() (in module audioop), 1030
getsampwidth() (aifc.aifc method), 1034
getsampwidth() (sunau.AU_read method), 1037
getsampwidth() (wave.Wave_read method), 1039
getscreen() (in module turtle), 1129
getscrollbarvalues() (FrameWork.ScrolledWindow method), 1413
getsequences() (mhlib.Folder method), 827
getsequencesfilename() (mhlib.Folder method), 827
getservbyname() (in module socket), 717
getservbyport() (in module socket), 717
GetSetDescriptorType (in module types), 215
getshapes() (in module turtle), 1134
getsid() (in module os), 419
getsignal() (in module signal), 752
getsitepackages() (in module site), 1291
getsize() (chunk.Chunk method), 1041
getsize() (in module os.path), 293
getsizeof() (in module sys), 1252
getsizes() (in module imgfile), 1441
getslice() (in module operator), 285
getsockname() (socket.socket method), 720
getsockopt() (socket.socket method), 720
getsource() (in module inspect), 1287
getsourcefile() (in module inspect), 1287
getsourcelines() (in module inspect), 1287
getspall() (in module spwd), 1385
getspnam() (in module spwd), 1385
getstate() (in module random), 263
getstatus() (in module commands), 1401
getstatusoutput() (in module commands), 1401
getstr() (curses.window method), 563
GetString() (msilib.Record method), 1367
getSubject() (logging.handlers.SMTPHandler method), 552
getsubtype() (mimetools.Message method), 829
GetSummaryInformation() (msilib.Database method), 1366
getSystemId() (xml.sax.xmlreader.InputSource method), 899
getSystemId() (xml.sax.xmlreader.Locator method), 898
getsyx() (in module curses), 556
gettarinfo() (tarfile.TarFile method), 381
gettempdir() (in module tempfile), 306
gettempprefix() (in module tempfile), 307
getTestCaseNames() (unittest.TestLoader method), 1195
gettext (module), 1049
gettext() (gettext.GNUTranslations method), 1054
gettext() (gettext.NullTranslations method), 1052
gettext() (in module gettext), 1050
gettext() (in module locale), 1065
GetTicks() (in module MacOS), 1405
gettimeout() (socket.socket method), 722
gettrace() (in module sys), 1252
getturtle() (in module turtle), 1128
gettype() (mimetools.Message method), 829

getType() (xml.sax.xmlreader.Attributes method), 899
getuid() (in module os), 418
geturl() (urlparse.ParseResult method), 987
getuser() (in module getpass), 554
getuserbase() (in module site), 1291
getusersitepackages() (in module site), 1291
getvalue() (io.BytesIO method), 451
getvalue() (io.StringIO method), 455
getvalue() (StringIO.StringIO method), 120
getValue() (xml.sax.xmlreader.Attributes method), 900
getValueByQName() (xml.sax.xmlreader.AttributesNS method), 900
getwch() (in module msvcrt), 1371
getwche() (in module msvcrt), 1371
getweakrefcount() (in module weakref), 207
getweakrefs() (in module weakref), 208
getwelcome() (ftplib.FTP method), 958
getwelcome() (nntplib.NNTP method), 970
getwelcome() (poplib.POP3 method), 961
getwin() (in module curses), 556
getwindowsversion() (in module sys), 1253
getwriter() (in module codecs), 126
getyx() (curses.window method), 563
gid (tarfile.TarInfo attribute), 383
GIL, 1454
GL (module), 1441
gl (module), 1439
glob
    module, 308
glob (module), 307
glob() (in module glob), 307
glob() (msilib.Directory method), 1369
global interpreter lock, 1454
globals() (built-in function), 11
globs (doctest.DocTest attribute), 1169
gmtime() (in module time), 458
gname (tarfile.TarInfo attribute), 383
GNOME, 1055
GNU_FORMAT (in module tarfile), 378
gnu_getopt() (in module getopt), 520
got (doctest.DocTestFailure attribute), 1176
goto() (in module turtle), 1114
Graphical User Interface, 1075
GREATER (in module token), 1333
GREATEREQUAL (in module token), 1333
Greenwich Mean Time, 456
grey22grey() (in module imageop), 1033
grey2grey2() (in module imageop), 1033
grey2grey4() (in module imageop), 1033
grey2mono() (in module imageop), 1032
grey42grey() (in module imageop), 1033
group() (nntplib.NNTP method), 971
group() (re.MatchObject method), 98
groupby() (in module itertools), 270
groupdict() (re.MatchObject method), 100
groupindex (re.RegexObject attribute), 98
groups (re.RegexObject attribute), 98
groups() (re.MatchObject method), 99
grp (module), 1385
gt() (in module operator), 282
guess_all_extensions() (in module mimetypes), 829
guess_all_extensions() (mimetypes.MimeTypes method), 831
guess_extension() (in module mimetypes), 829
guess_extension() (mimetypes.MimeTypes method), 831
guess_scheme() (in module wsgiref.util), 922
guess_type() (in module mimetypes), 829
guess_type() (mimetypes.MimeTypes method), 831
GUI, 1075
gzip (module), 366
GzipFile (class in gzip), 367

H

halfdelay() (in module curses), 556
handle() (BaseHTTPServer.BaseHTTPRequestHandler method), 998
handle() (logging.Handler method), 526
handle() (logging.Logger method), 525
handle() (logging.NullHandler method), 546
handle() (SocketServer.BaseRequestHandler method), 992
handle() (wsgiref.simple_server.WSGIRequestHandler method), 926
handle_accept() (asyncore.dispatcher method), 758
handle_charref() (HTMLParser.HTMLParser method), 849
handle_charref() (sgmllib.SGMLParser method), 853
handle_close() (asyncore.dispatcher method), 757
handle_comment() (HTMLParser.HTMLParser method), 849
handle_comment() (sgmllib.SGMLParser method), 854
handle_connect() (asyncore.dispatcher method), 757
handle_data() (HTMLParser.HTMLParser method), 849
handle_data() (sgmllib.SGMLParser method), 853
handle_decl() (HTMLParser.HTMLParser method), 849
handle_decl() (sgmllib.SGMLParser method), 854
handle_endtag() (HTMLParser.HTMLParser method), 849
handle_endtag() (sgmllib.SGMLParser method), 853
handle_entityref() (HTMLParser.HTMLParser method), 849

handle_entityref() (sgmllib.SGMLParser method), 854
handle_error() (asyncore.dispatcher method), 757
handle_error() (SocketServer.BaseServer method), 991
handle_expt() (asyncore.dispatcher method), 757
handle_image() (htmllib.HTMLParser method), 856
handle_one_request() (BaseHTTPServer.BaseHTTPRequestHandler method), 998
handle_pi() (HTMLParser.HTMLParser method), 850
handle_read() (asyncore.dispatcher method), 757
handle_request() (SimpleXMLRPCServer.CGIXMLRPCRequestHandler method), 1027
handle_request() (SocketServer.BaseServer method), 990
handle_startendtag() (HTMLParser.HTMLParser method), 849
handle_starttag() (HTMLParser.HTMLParser method), 849
handle_starttag() (sgmllib.SGMLParser method), 853
handle_timeout() (SocketServer.BaseServer method), 991
handle_write() (asyncore.dispatcher method), 757
handleError() (logging.Handler method), 526
handleError() (logging.handlers.SocketHandler method), 548
handler() (in module cgitb), 921
HAS_ALPN (in module ssl), 735
has_children() (symtable.SymbolTable method), 1331
has_colors() (in module curses), 556
has_data() (urllib2.Request method), 941
HAS_ECDH (in module ssl), 735
has_exec() (symtable.SymbolTable method), 1331
has_extn() (smtplib.SMTP method), 975
has_header() (csv.Sniffer method), 390
has_header() (urllib2.Request method), 942
has_ic() (in module curses), 556
has_il() (in module curses), 556
has_import_star() (symtable.SymbolTable method), 1331
has_ipv6 (in module socket), 715
has_key (2to3 fixer), 1205
has_key() (bsddb.bsddbobject method), 339
has_key() (dict method), 54
has_key() (email.message.Message method), 768
has_key() (in module curses), 556
has_key() (mailbox.Mailbox method), 808
has_nonstandard_attr() (cookielib.Cookie method), 1010
HAS_NPN (in module ssl), 736
has_option() (ConfigParser.RawConfigParser method), 398
has_option() (optparse.OptionParser method), 510
has_section() (ConfigParser.RawConfigParser method), 397
HAS_SNI (in module ssl), 735
HAS_TLSv1_3 (in module ssl), 736
hasattr() (built-in function), 12
hasAttribute() (xml.dom.Element method), 878
hasAttributeNS() (xml.dom.Element method), 878
hasAttributes() (xml.dom.Node method), 875
hasChildNodes() (xml.dom.Node method), 876
hascompare (in module dis), 1342
hasconst (in module dis), 1342
hasFeature() (xml.dom.DOMImplementation method), 874
hasfree (in module dis), 1342
hash() (built-in function), 12
hash.block_size (in module hashlib), 410
hash.digest_size (in module hashlib), 410
hashable, 1454
Hashable (class in collections), 187
hashlib (module), 409
hashlib.algorithms (in module hashlib), 410
hashopen() (in module bsddb), 339
hasjabs (in module dis), 1342
hasjrel (in module dis), 1342
haslocal (in module dis), 1342
hasname (in module dis), 1342
HAVE_ARGUMENT (opcode), 1350
have_unicode (in module test.support), 1210
head() (nntplib.NNTP method), 971
Header (class in email.header), 779
header_encode() (email.charset.Charset method), 782
header_encoding (email.charset.Charset attribute), 781
header_items() (urllib2.Request method), 942
header_offset (zipfile.ZipInfo attribute), 376
HeaderError, 378
HeaderParseError, 784
headers
    MIME, 829, 914
headers (BaseHTTPServer.BaseHTTPRequestHandler attribute), 997
Headers (class in wsgiref.headers), 924
headers (rfc822.Message attribute), 839
headers (xmlrpclib.ProtocolError attribute), 1020
heading() (in module turtle), 1119
heading() (ttk.Treeview method), 1098
heapify() (in module heapq), 189
heapmin() (in module msvcrt), 1372
heappop() (in module heapq), 189
heappush() (in module heapq), 189

heappushpop() (in module heapq), 189
heapq (module), 189
heapreplace() (in module heapq), 190
helo() (smtplib.SMTP method), 974
help
    online, 1151
help (optparse.Option attribute), 506
help() (built-in function), 12
help() (nntplib.NNTP method), 971
herror, 714
hex (uuid.UUID attribute), 982
hex() (built-in function), 12
hex() (float method), 36
hex() (in module future_builtins), 1264
hexadecimal
    literals, 33
hexbin() (in module binhex), 842
hexdigest() (hashlib.hash method), 410
hexdigest() (hmac.HMAC method), 412
hexdigest() (md5.md5 method), 413
hexdigest() (sha.sha method), 414
hexdigits (in module string), 75
hexlify() (in module binascii), 844
hexversion (in module sys), 1253
hidden() (curses.panel.Panel method), 576
hide() (curses.panel.Panel method), 576
hide() (ttk.Notebook method), 1093
hide_cookie2 (cookielib.CookiePolicy attribute), 1007
hide_form() (fl.form method), 1435
hideturtle() (in module turtle), 1124
HierarchyRequestErr, 881
HIGHEST_PROTOCOL (in module pickle), 319
HKEY_CLASSES_ROOT (in module _winreg), 1377
HKEY_CURRENT_CONFIG (in module _winreg), 1378
HKEY_CURRENT_USER (in module _winreg), 1377
HKEY_DYN_DATA (in module _winreg), 1378
HKEY_LOCAL_MACHINE (in module _winreg), 1377
HKEY_PERFORMANCE_DATA (in module _winreg), 1378
HKEY_USERS (in module _winreg), 1378
hline() (curses.window method), 563
HList (class in Tix), 1107
hls_to_rgb() (in module colorsys), 1042
hmac (module), 411
HOME, 292, 1292
home() (in module turtle), 1115
HOMEDRIVE, 292
HOMEPATH, 292
hook_compressed() (in module fileinput), 297
hook_encoded() (in module fileinput), 297
hosts (netrc.netrc attribute), 403
hotshot (module), 1233
hotshot.stats (module), 1234
hour (datetime.datetime attribute), 154
hour (datetime.time attribute), 160
HRESULT (class in ctypes), 617
hStdError (subprocess.STARTUPINFO attribute), 709
hStdInput (subprocess.STARTUPINFO attribute), 709
hStdOutput (subprocess.STARTUPINFO attribute), 709
hsv_to_rgb() (in module colorsys), 1042
ht() (in module turtle), 1124
HTML, 847, 855, 936
HTMLCalendar (class in calendar), 172
HtmlDiff (class in difflib), 110
HtmlDiff.__init__() (in module difflib), 110
HtmlDiff.make_file() (in module difflib), 111
HtmlDiff.make_table() (in module difflib), 111
htmlentitydefs (module), 856
htmllib
    module, 936
htmllib (module), 855
HTMLParseError, 847, 855
HTMLParser (class in htmllib), 855, 1359
HTMLParser (class in HTMLParser), 847
HTMLParser (module), 847
htonl() (in module socket), 718
htons() (in module socket), 718
HTTP
    httplib (standard module), 949
    protocol, 914, 936, 949, 996
http_error_301() (urllib2.HTTPRedirectHandler method), 945
http_error_302() (urllib2.HTTPRedirectHandler method), 945
http_error_303() (urllib2.HTTPRedirectHandler method), 945
http_error_307() (urllib2.HTTPRedirectHandler method), 945
http_error_401() (urllib2.HTTPBasicAuthHandler method), 946
http_error_401() (urllib2.HTTPDigestAuthHandler method), 946
http_error_407() (urllib2.ProxyBasicAuthHandler method), 946
http_error_407() (urllib2.ProxyDigestAuthHandler method), 946
http_error_auth_reqed() (urllib2.AbstractBasicAuthHandler method), 946
http_error_auth_reqed()

(urllib2.AbstractDigestAuthHandler method), 946
http_error_default() (urllib2.BaseHandler method), 944
http_error_nnn() (urllib2.BaseHandler method), 944
http_open() (urllib2.HTTPHandler method), 947
HTTP_PORT (in module httplib), 951
http_proxy, 931, 938, 948
http_response() (urllib2.HTTPErrorProcessor method), 947
http_version (wsgiref.handlers.BaseHandler attribute), 930
HTTPBasicAuthHandler (class in urllib2), 940
HTTPConnection (class in httplib), 949
HTTPCookieProcessor (class in urllib2), 940
httpd, 996
HTTPDefaultErrorHandler (class in urllib2), 940
HTTPDigestAuthHandler (class in urllib2), 940
HTTPError, 939
HTTPErrorProcessor (class in urllib2), 941
HTTPException, 950
HTTPHandler (class in logging.handlers), 553
HTTPHandler (class in urllib2), 941
httplib (module), 949
HTTPMessage (class in httplib), 950
HTTPPasswordMgr (class in urllib2), 940
HTTPPasswordMgrWithDefaultRealm (class in urllib2), 940
HTTPRedirectHandler (class in urllib2), 940
HTTPResponse (class in httplib), 950
https_open() (urllib2.HTTPSHandler method), 947
HTTPS_PORT (in module httplib), 951
https_response() (urllib2.HTTPErrorProcessor method), 947
HTTPSConnection (class in httplib), 950
HTTPServer (class in BaseHTTPServer), 997
HTTPSHandler (class in urllib2), 941
hypertext, 855
hypot() (in module math), 229

I

I (in module re), 94
I/O control
  buffering, 15, 420, 721
  POSIX, 1388
  tty, 1388
  UNIX, 1390
iadd() (in module operator), 285
iand() (in module operator), 286
IC (class in ic), 1403
ic (module), 1403
icglue
  module, 1403
iconcat() (in module operator), 286
icopen (module), 1448
id() (built-in function), 12
id() (unittest.TestCase method), 1193
idcok() (curses.window method), 563
ident (in module cd), 1430
ident (select.kevent attribute), 625
ident (threading.Thread attribute), 630
identchars (cmd.Cmd attribute), 1069
identify() (ttk.Notebook method), 1093
identify() (ttk.Treeview method), 1099
identify() (ttk.Widget method), 1090
identify_column() (ttk.Treeview method), 1099
identify_element() (ttk.Treeview method), 1099
identify_region() (ttk.Treeview method), 1099
identify_row() (ttk.Treeview method), 1099
idioms (2to3 fixer), 1205
idiv() (in module operator), 286
IDLE, 1455
IDLE, 1140
idle() (FrameWork.Application method), 1412
IDLESTARTUP, 1146
idlok() (curses.window method), 564
IEEE-754, 1293
if
  statement, 31
ifilter() (in module itertools), 271
ifilterfalse() (in module itertools), 271
ifloordiv() (in module operator), 286
iglob() (in module glob), 307
ignorableWhitespace() (xml.sax.handler.ContentHandler method), 894
ignore_errors() (in module codecs), 127
IGNORE_EXCEPTION_DETAIL (in module doctest), 1160
ignore_patterns() (in module shutil), 310
IGNORECASE (in module re), 94
ihave() (nntplib.NNTP method), 972
ilshift() (in module operator), 286
imag (numbers.Complex attribute), 223
imageop (module), 1032
imap() (in module itertools), 272
imap() (multiprocessing.pool.multiprocessing.Pool method), 664
IMAP4
  protocol, 962
IMAP4 (class in imaplib), 963
IMAP4.abort, 963
IMAP4.error, 963
IMAP4.readonly, 963
IMAP4_SSL
  protocol, 962
IMAP4_SSL (class in imaplib), 963
IMAP4_stream

protocol, 962
IMAP4_stream (class in imaplib), 963
imap_unordered() (multiprocessing.pool.multiprocessing.Pool method), 664
imaplib (module), 962
imgfile (module), 1441
imghdr (module), 1042
immedok() (curses.window method), 564
immutable, 1455
ImmutableSet (class in sets), 199
imod() (in module operator), 286
imp
module, 24
imp (module), 1305
ImpImporter (class in pkgutil), 1315
ImpLoader (class in pkgutil), 1315
import
statement, 24, 1305, 1309
import (2to3 fixer), 1205
import_file() (imputil.DynLoadSuffixImporter method), 1310
import_fresh_module() (in module test.support), 1212
IMPORT_FROM (opcode), 1348
import_module() (in module importlib), 1309
import_module() (in module test.support), 1212
IMPORT_NAME (opcode), 1348
IMPORT_STAR (opcode), 1346
import_top() (imputil.Importer method), 1309
importer, 1455
Importer (class in imputil), 1309
ImportError, 69
importing, 1455
importlib (module), 1308
ImportManager (class in imputil), 1309
imports (2to3 fixer), 1205
imports2 (2to3 fixer), 1205
ImportWarning, 72
ImproperConnectionState, 951
imputil (module), 1309
imul() (in module operator), 286
in
operator, 33, 38
in_dll() (ctypes._CData method), 615
in_table_a1() (in module stringprep), 141
in_table_b1() (in module stringprep), 141
in_table_c11() (in module stringprep), 142
in_table_c11_c12() (in module stringprep), 142
in_table_c12() (in module stringprep), 142
in_table_c21() (in module stringprep), 142
in_table_c21_c22() (in module stringprep), 142
in_table_c22() (in module stringprep), 142
in_table_c3() (in module stringprep), 142
in_table_c4() (in module stringprep), 142
in_table_c5() (in module stringprep), 142
in_table_c6() (in module stringprep), 142
in_table_c7() (in module stringprep), 142
in_table_c8() (in module stringprep), 142
in_table_c9() (in module stringprep), 142
in_table_d1() (in module stringprep), 142
in_table_d2() (in module stringprep), 142
inc() (EasyDialogs.ProgressBar method), 1410
inch() (curses.window method), 564
Incomplete, 844
IncompleteRead, 951
increment_lineno() (in module ast), 1329
IncrementalDecoder (class in codecs), 130
IncrementalEncoder (class in codecs), 129
IncrementalNewlineDecoder (class in io), 455
IncrementalParser (class in xml.sax.xmlreader), 896
indent (doctest.Example attribute), 1169
INDENT (in module token), 1333
IndentationError, 70
Independent JPEG Group, 1442
index (in module cd), 1430
index() (array.array method), 197
index() (in module operator), 283
index() (in module string), 86
index() (list method), 48
index() (str method), 41
index() (ttk.Notebook method), 1093
index() (ttk.Treeview method), 1099
IndexError, 69
indexOf() (in module operator), 285
IndexSizeErr, 881
inet_aton() (in module socket), 718
inet_ntoa() (in module socket), 718
inet_ntop() (in module socket), 719
inet_pton() (in module socket), 718
Inexact (class in decimal), 252
infile (shlex.shlex attribute), 1072
Infinity, 11, 86
info() (gettext.NullTranslations method), 1053
info() (in module logging), 532
info() (logging.Logger method), 524
infolist() (zipfile.ZipFile method), 372
InfoScrap() (in module Carbon.Scrap), 1417
ini file, 395
init() (in module fm), 1438
init() (in module mimetypes), 830
init_builtin() (in module imp), 1307
init_color() (in module curses), 557
init_database() (in module msilib), 1365
init_frozen() (in module imp), 1307
init_pair() (in module curses), 557
inited (in module mimetypes), 830
initgroups() (in module os), 417

initial_indent (textwrap.TextWrapper attribute), 123
initscr() (in module curses), 557
INPLACE_ADD (opcode), 1344
INPLACE_AND (opcode), 1344
INPLACE_DIVIDE (opcode), 1344
INPLACE_FLOOR_DIVIDE (opcode), 1344
INPLACE_LSHIFT (opcode), 1344
INPLACE_MODULO (opcode), 1344
INPLACE_MULTIPLY (opcode), 1344
INPLACE_OR (opcode), 1345
INPLACE_POWER (opcode), 1344
INPLACE_RSHIFT (opcode), 1344
INPLACE_SUBTRACT (opcode), 1344
INPLACE_TRUE_DIVIDE (opcode), 1344
INPLACE_XOR (opcode), 1344
input
    built-in function, 1258
input (2to3 fixer), 1205
input() (built-in function), 12
input() (in module fileinput), 295
input_charset (email.charset.Charset attribute), 781
input_codec (email.charset.Charset attribute), 782
InputOnly (class in Tix), 1108
InputSource (class in xml.sax.xmlreader), 896
InputType (in module cStringIO), 121
insch() (curses.window method), 564
insdelln() (curses.window method), 564
insert() (array.array method), 197
insert() (list method), 48
insert() (ttk.Notebook method), 1093
insert() (ttk.Treeview method), 1099
insert() (xml.etree.ElementTree.Element method), 868
insert_text() (in module readline), 696
insertBefore() (xml.dom.Node method), 876
InsertionLoc (class in aetypes), 1423
insertln() (curses.window method), 564
insnstr() (curses.window method), 564
insort() (in module bisect), 194
insort_left() (in module bisect), 193
insort_right() (in module bisect), 194
inspect (module), 1283
insstr() (curses.window method), 564
install() (gettext.NullTranslations method), 1053
install() (imputil.ImportManager method), 1309
install() (in module gettext), 1051
install_opener() (in module urllib2), 938
installaehandler() (MiniAEFrame.AEServer method), 1424
installAutoGIL() (in module autoGIL), 1414
installHandler() (in module unittest), 1202
instance() (in module new), 216
instancemethod() (in module new), 216
InstanceType (in module types), 214
instate() (ttk.Widget method), 1090
instr() (curses.window method), 564
instream (shlex.shlex attribute), 1072
int
    built-in function, 33
int (built-in class), 12
int (uuid.UUID attribute), 982
Int2AP() (in module imaplib), 963
integer
    division, 34
    division, long, 34
    literals, 33
    literals, long, 33
    object, 33
    types, operations on, 35
integer division, 1455
Integral (class in numbers), 224
Integrated Development Environment, 1140
Intel/DVI ADPCM, 1029
interact() (code.InteractiveConsole method), 1296
interact() (in module code), 1295
interact() (telnetlib.Telnet method), 980
interactive, 1455
InteractiveConsole (class in code), 1295
InteractiveInterpreter (class in code), 1295
intern (2to3 fixer), 1206
intern() (built-in function), 27
internal_attr (zipfile.ZipInfo attribute), 375
Internaldate2tuple() (in module imaplib), 963
internalSubset (xml.dom.DocumentType attribute), 877
Internet, 911
Internet Config, 932
interpolation, string (%), 46
InterpolationDepthError, 397
InterpolationError, 397
InterpolationMissingOptionError, 397
InterpolationSyntaxError, 397
interpreted, 1455
interpreter prompts, 1256
interrupt() (sqlite3.Connection method), 347
interrupt_main() (in module thread), 637
intersection() (frozenset method), 51
intersection_update() (frozenset method), 52
IntlText (class in aetypes), 1423
IntlWritingCode (class in aetypes), 1423
intro (cmd.Cmd attribute), 1069
IntType (in module types), 214
InuseAttributeErr, 881
inv() (in module operator), 283
InvalidAccessErr, 881
InvalidCharacterErr, 881
InvalidModificationErr, 881

InvalidOperation (class in decimal), 252
InvalidStateErr, 881
InvalidURL, 951
invert() (in module operator), 283
io (module), 445
IOBase (class in io), 447
ioctl() (in module fcntl), 1391
ioctl() (socket.socket method), 720
IOError, 68
ior() (in module operator), 286
ipow() (in module operator), 286
irepeat() (in module operator), 286
IRIS Font Manager, 1438
IRIX
    threads, 638
irshift() (in module operator), 287
is
    operator, 32
is not
    operator, 32
is_() (in module operator), 283
is_alive() (multiprocessing.Process method), 645
is_alive() (threading.Thread method), 630
is_assigned() (symtable.Symbol method), 1332
is_blocked() (cookielib.DefaultCookiePolicy method), 1008
is_builtin() (in module imp), 1307
is_canonical() (decimal.Context method), 249
is_canonical() (decimal.Decimal method), 241
IS_CHARACTER_JUNK() (in module difflib), 114
is_data() (multifile.MultiFile method), 835
is_declared_global() (symtable.Symbol method), 1332
is_empty() (asynchat.fifo method), 762
is_expired() (cookielib.Cookie method), 1010
is_finite() (decimal.Context method), 249
is_finite() (decimal.Decimal method), 241
is_free() (symtable.Symbol method), 1332
is_frozen() (in module imp), 1307
is_global() (symtable.Symbol method), 1332
is_hop_by_hop() (in module wsgiref.util), 923
is_imported() (symtable.Symbol method), 1332
is_infinite() (decimal.Context method), 249
is_infinite() (decimal.Decimal method), 241
is_integer() (float method), 36
is_jython (in module test.support), 1210
IS_LINE_JUNK() (in module difflib), 114
is_linetouched() (curses.window method), 564
is_local() (symtable.Symbol method), 1332
is_multipart() (email.message.Message method), 766
is_namespace() (symtable.Symbol method), 1332
is_nan() (decimal.Context method), 249
is_nan() (decimal.Decimal method), 241
is_nested() (symtable.SymbolTable method), 1331
is_normal() (decimal.Context method), 249
is_normal() (decimal.Decimal method), 241
is_not() (in module operator), 283
is_not_allowed() (cookielib.DefaultCookiePolicy method), 1008
is_optimized() (symtable.SymbolTable method), 1331
is_package() (zipimport.zipimporter method), 1313
is_parameter() (symtable.Symbol method), 1332
is_python_build() (in module sysconfig), 1262
is_qnan() (decimal.Context method), 249
is_qnan() (decimal.Decimal method), 241
is_referenced() (symtable.Symbol method), 1332
is_resource_enabled() (in module test.support), 1210
is_scriptable() (in module gensuitemodule), 1420
is_set() (threading.Event method), 635
is_signed() (decimal.Context method), 249
is_signed() (decimal.Decimal method), 241
is_snan() (decimal.Context method), 249
is_snan() (decimal.Decimal method), 242
is_subnormal() (decimal.Context method), 249
is_subnormal() (decimal.Decimal method), 242
is_tarfile() (in module tarfile), 378
is_term_resized() (in module curses), 557
is_tracked() (in module gc), 1282
is_unverifiable() (urllib2.Request method), 942
is_wintouched() (curses.window method), 564
is_zero() (decimal.Context method), 249
is_zero() (decimal.Decimal method), 242
is_zipfile() (in module zipfile), 371
isabs() (in module os.path), 293
isabstract() (in module inspect), 1286
isAlive() (threading.Thread method), 630
isalnum() (in module curses.ascii), 574
isalnum() (str method), 41
isalpha() (in module curses.ascii), 574
isalpha() (str method), 41
isascii() (in module curses.ascii), 574
isatty() (chunk.Chunk method), 1041
isatty() (file method), 58
isatty() (in module os), 423
isatty() (io.IOBase method), 448
isblank() (in module curses.ascii), 574
isblk() (tarfile.TarInfo method), 383
isbuiltin() (in module inspect), 1286
isCallable() (in module operator), 287
ischr() (tarfile.TarInfo method), 383
isclass() (in module inspect), 1285
iscntrl() (in module curses.ascii), 574
iscode() (in module inspect), 1286
iscomment() (rfc822.Message method), 838
isctrl() (in module curses.ascii), 574
isDaemon() (threading.Thread method), 631

isdatadescriptor() (in module inspect), 1286
isdecimal() (unicode method), 46
isdev() (tarfile.TarInfo method), 383
isdigit() (in module curses.ascii), 574
isdigit() (str method), 41
isdir() (in module os.path), 293
isdir() (tarfile.TarInfo method), 383
isdisjoint() (frozenset method), 51
isdown() (in module turtle), 1121
iselement() (in module xml.etree.ElementTree), 865
isenabled() (in module gc), 1281
isEnabledFor() (logging.Logger method), 523
isendwin() (in module curses), 557
ISEOF() (in module token), 1333
isexpr() (in module parser), 1323
isexpr() (parser.ST method), 1324
isfifo() (tarfile.TarInfo method), 383
isfile() (in module os.path), 293
isfile() (tarfile.TarInfo method), 383
isfirstline() (in module fileinput), 296
isframe() (in module inspect), 1286
isfunction() (in module inspect), 1285
isgenerator() (in module inspect), 1285
isgeneratorfunction() (in module inspect), 1285
isgetsetdescriptor() (in module inspect), 1286
isgraph() (in module curses.ascii), 574
isheader() (rfc822.Message method), 838
isinf() (in module cmath), 233
isinf() (in module math), 227
isinstance (2to3 fixer), 1206
isinstance() (built-in function), 13
iskeyword() (in module keyword), 1334
islast() (rfc822.Message method), 838
isleap() (in module calendar), 173
islice() (in module itertools), 272
islink() (in module os.path), 293
islnk() (tarfile.TarInfo method), 383
islower() (in module curses.ascii), 574
islower() (str method), 41
isMappingType() (in module operator), 287
ismemberdescriptor() (in module inspect), 1286
ismeta() (in module curses.ascii), 574
ismethod() (in module inspect), 1285
ismethoddescriptor() (in module inspect), 1286
ismodule() (in module inspect), 1285
ismount() (in module os.path), 293
isnan() (in module cmath), 233
isnan() (in module math), 227
ISNONTERMINAL() (in module token), 1333
isNumberType() (in module operator), 287
isnumeric() (unicode method), 46
isocalendar() (datetime.date method), 151
isocalendar() (datetime.datetime method), 157
isoformat() (datetime.date method), 151
isoformat() (datetime.datetime method), 157
isoformat() (datetime.time method), 160
isolation_level (sqlite3.Connection attribute), 345
isoweekday() (datetime.date method), 151
isoweekday() (datetime.datetime method), 156
isprint() (in module curses.ascii), 574
ispunct() (in module curses.ascii), 574
isqueued() (in module fl), 1434
isreadable() (in module pprint), 219
isreadable() (pprint.PrettyPrinter method), 219
isrecursive() (in module pprint), 219
isrecursive() (pprint.PrettyPrinter method), 219
isreg() (tarfile.TarInfo method), 383
isReservedKey() (Cookie.Morsel method), 1013
isroutine() (in module inspect), 1286
isSameNode() (xml.dom.Node method), 876
isSequenceType() (in module operator), 287
isSet() (threading.Event method), 635
isspace() (in module curses.ascii), 574
isspace() (str method), 41
isstdin() (in module fileinput), 296
issubclass() (built-in function), 13
issubset() (frozenset method), 51
issuite() (in module parser), 1324
issuite() (parser.ST method), 1324
issuperset() (frozenset method), 51
issym() (tarfile.TarInfo method), 383
ISTERMINAL() (in module token), 1333
istitle() (str method), 41
istraceback() (in module inspect), 1285
isub() (in module operator), 287
isupper() (in module curses.ascii), 574
isupper() (str method), 42
isvisible() (in module turtle), 1125
isxdigit() (in module curses.ascii), 574
item() (ttk.Treeview method), 1099
item() (xml.dom.NamedNodeMap method), 879
item() (xml.dom.NodeList method), 876
itemgetter() (in module operator), 288
items() (ConfigParser.ConfigParser method), 399
items() (ConfigParser.RawConfigParser method), 398
items() (dict method), 54
items() (email.message.Message method), 768
items() (mailbox.Mailbox method), 808
items() (xml.etree.ElementTree.Element method), 867
itemsize (array.array attribute), 196
itemsize (memoryview attribute), 62
ItemsView (class in collections), 188
iter() (built-in function), 13
iter() (xml.etree.ElementTree.Element method), 868
iter() (xml.etree.ElementTree.ElementTree method), 869

iter_child_nodes() (in module ast), 1329
iter_fields() (in module ast), 1329
iter_importers() (in module pkgutil), 1315
iter_modules() (in module pkgutil), 1316
iterable, 1455
Iterable (class in collections), 187
IterableUserDict (class in UserDict), 211
iterator, 1455
Iterator (class in collections), 187
iterator protocol, 37
iterdecode() (in module codecs), 127
iterdump (sqlite3.Connection attribute), 350
iterencode() (in module codecs), 127
iterencode() (json.JSONEncoder method), 803
iterfind() (xml.etree.ElementTree.Element method), 868
iterfind() (xml.etree.ElementTree.ElementTree method), 869
iteritems() (dict method), 55
iteritems() (mailbox.Mailbox method), 808
iterkeyrefs() (weakref.WeakKeyDictionary method), 208
iterkeys() (dict method), 55
iterkeys() (mailbox.Mailbox method), 807
itermonthdates() (calendar.Calendar method), 171
itermonthdays() (calendar.Calendar method), 171
itermonthdays2() (calendar.Calendar method), 171
iterparse() (in module xml.etree.ElementTree), 866
itertext() (xml.etree.ElementTree.Element method), 868
itertools (2to3 fixer), 1206
itertools (module), 266
itertools_imports (2to3 fixer), 1206
itervaluerefs() (weakref.WeakValueDictionary method), 208
itervalues() (dict method), 55
itervalues() (mailbox.Mailbox method), 807
iterweekdays() (calendar.Calendar method), 170
ITIMER_PROF (in module signal), 752
ITIMER_REAL (in module signal), 752
ITIMER_VIRTUAL (in module signal), 752
ItimerError, 752
itruediv() (in module operator), 287
ixor() (in module operator), 287
izip() (in module itertools), 273
izip_longest() (in module itertools), 273

J

Jansen, Jack, 845
java_ver() (in module platform), 578
JFIF, 1442
join() (in module os.path), 293
join() (in module string), 87
join() (multiprocessing.JoinableQueue method), 649
join() (multiprocessing.pool.multiprocessing.Pool method), 664
join() (multiprocessing.Process method), 645
join() (Queue.Queue method), 206
join() (str method), 42
join() (threading.Thread method), 630
join_thread() (multiprocessing.Queue method), 648
JoinableQueue (class in multiprocessing), 649
joinfields() (in module string), 87
jpeg (module), 1442
js_output() (Cookie.BaseCookie method), 1012
js_output() (Cookie.Morsel method), 1013
json (module), 797
JSONDecoder (class in json), 801
JSONEncoder (class in json), 802
JUMP_ABSOLUTE (opcode), 1348
JUMP_FORWARD (opcode), 1348
JUMP_IF_FALSE_OR_POP (opcode), 1348
JUMP_IF_TRUE_OR_POP (opcode), 1348
jumpahead() (in module random), 263

K

kbhit() (in module msvcrt), 1371
KDEDIR, 913
kevent() (in module select), 621
key (Cookie.Morsel attribute), 1013
key function, 1456
KEY_ALL_ACCESS (in module _winreg), 1378
KEY_CREATE_LINK (in module _winreg), 1378
KEY_CREATE_SUB_KEY (in module _winreg), 1378
KEY_ENUMERATE_SUB_KEYS (in module _winreg), 1378
KEY_EXECUTE (in module _winreg), 1378
KEY_NOTIFY (in module _winreg), 1378
KEY_QUERY_VALUE (in module _winreg), 1378
KEY_READ (in module _winreg), 1378
KEY_SET_VALUE (in module _winreg), 1378
KEY_WOW64_32KEY (in module _winreg), 1379
KEY_WOW64_64KEY (in module _winreg), 1378
KEY_WRITE (in module _winreg), 1378
KeyboardInterrupt, 69
KeyError, 69
keyname() (in module curses), 557
keypad() (curses.window method), 564
keyrefs() (weakref.WeakKeyDictionary method), 208
keys() (bsddb.bsddbobject method), 339
keys() (dict method), 55
keys() (email.message.Message method), 768
keys() (mailbox.Mailbox method), 807
keys() (sqlite3.Row method), 354
keys() (xml.etree.ElementTree.Element method), 868
keysubst() (in module aetools), 1421
KeysView (class in collections), 188

Keyword (class in aetypes), 1423
keyword (module), 1334
keyword argument, 1456
keywords (functools.partial attribute), 282
kill() (in module os), 438
kill() (subprocess.Popen method), 708
killchar() (in module curses), 557
killpg() (in module os), 438
knee
    module, 1308, 1312
knownfiles (in module mimetypes), 830
kqueue() (in module select), 621
kwlist (in module keyword), 1335

L

L (in module re), 94
label() (EasyDialogs.ProgressBar method), 1409
LabelEntry (class in Tix), 1105
LabelFrame (class in Tix), 1106
lambda, 1456
LambdaType (in module types), 214
LANG, 1049, 1051, 1059, 1062
LANGUAGE, 1049, 1051
language
    C, 33
large files, 1383
LargeZipFile, 371
last (multifile.MultiFile attribute), 835
last() (bsddb.bsddbobject method), 340
last() (dbhash.dbhash method), 338
last() (nntplib.NNTP method), 971
last_accepted (multiprocessing.connection.Listener attribute), 666
last_traceback (in module sys), 1254
last_type (in module sys), 1254
last_value (in module sys), 1254
lastChild (xml.dom.Node attribute), 875
lastcmd (cmd.Cmd attribute), 1069
lastgroup (re.MatchObject attribute), 100
lastindex (re.MatchObject attribute), 100
lastpart() (MimeWriter.MimeWriter method), 832
lastrowid (sqlite3.Cursor attribute), 353
launch() (in module findertools), 1407
launchurl() (ic.IC method), 1404
launchurl() (in module ic), 1403
layout() (ttk.Style method), 1102
LBRACE (in module token), 1333
LBYL, 1456
LC_ALL, 1049, 1051
LC_ALL (in module locale), 1064
LC_COLLATE (in module locale), 1064
LC_CTYPE (in module locale), 1064
LC_MESSAGES, 1049, 1051
LC_MESSAGES (in module locale), 1064
LC_MONETARY (in module locale), 1064
LC_NUMERIC (in module locale), 1064
LC_TIME (in module locale), 1064
lchflags() (in module os), 428
lchmod() (in module os), 428
lchown() (in module os), 428
ldexp() (in module math), 227
ldgettext() (in module gettext), 1050
ldngettext() (in module gettext), 1050
le() (in module operator), 282
leapdays() (in module calendar), 173
leaveok() (curses.window method), 564
left (filecmp.dircmp attribute), 303
left() (in module turtle), 1114
left_list (filecmp.dircmp attribute), 303
left_only (filecmp.dircmp attribute), 303
LEFTSHIFT (in module token), 1333
LEFTSHIFTEQUAL (in module token), 1333
len
    built-in function, 38, 53
len() (built-in function), 13
length (xml.dom.NamedNodeMap attribute), 879
length (xml.dom.NodeList attribute), 876
LESS (in module token), 1333
LESSEQUAL (in module token), 1333
letters (in module string), 75
level (multifile.MultiFile attribute), 835
lexists() (in module os.path), 292
lgamma() (in module math), 230
lgettext() (gettext.GNUTranslations method), 1054
lgettext() (gettext.NullTranslations method), 1052
lgettext() (in module gettext), 1050
lib2to3 (module), 1207
libc_ver() (in module platform), 579
library (in module dbm), 335
library (ssl.SSLError attribute), 726
LibraryLoader (class in ctypes), 607
license (built-in variable), 30
LifoQueue (class in Queue), 204
light-weight processes, 637
limit_denominator() (fractions.Fraction method), 261
lin2adpcm() (in module audioop), 1030
lin2alaw() (in module audioop), 1030
lin2lin() (in module audioop), 1030
lin2ulaw() (in module audioop), 1030
line() (msilib.Dialog method), 1370
line-buffered I/O, 15
line_buffering (io.TextIOWrapper attribute), 454
line_num (csv.csvreader attribute), 392
linecache (module), 309
lineno (ast.AST attribute), 1326
lineno (doctest.DocTest attribute), 1169
lineno (doctest.Example attribute), 1169

lineno (pyclbr.Class attribute), 1338
lineno (pyclbr.Function attribute), 1338
lineno (shlex.shlex attribute), 1072
lineno (xml.parsers.expat.ExpatError attribute), 906
lineno() (in module fileinput), 296
LINES, 556, 560
linesep (in module os), 444
lineterminator (csv.Dialect attribute), 391
link() (in module os), 428
linkmodel (in module MacOS), 1405
linkname (tarfile.TarInfo attribute), 383
linux_distribution() (in module platform), 579
list, 1456
list
    object, 37, 48
    type, operations on, 48
list comprehension, 1456
list() (imaplib.IMAP4 method), 965
list() (multiprocessing.managers.SyncManager method), 659
list() (nntplib.NNTP method), 970
list() (poplib.POP3 method), 961
list() (tarfile.TarFile method), 380
LIST_APPEND (opcode), 1346
list_dialects() (in module csv), 389
list_folders() (mailbox.Maildir method), 810
list_folders() (mailbox.MH method), 812
listallfolders() (mhlib.MH method), 826
listallsubfolders() (mhlib.MH method), 826
listdir() (in module dircache), 314
listdir() (in module os), 428
listen() (asyncore.dispatcher method), 758
listen() (in module logging.config), 536
listen() (in module turtle), 1132
listen() (socket.socket method), 721
Listener (class in multiprocessing.connection), 666
listfolders() (mhlib.MH method), 826
listmessages() (mhlib.Folder method), 827
listMethods() (xmlrpclib.ServerProxy.system method), 1017
ListNoteBook (class in Tix), 1107
listsubfolders() (mhlib.MH method), 826
ListType (in module types), 214
literal_eval() (in module ast), 1329
literals
    complex number, 33
    floating point, 33
    hexadecimal, 33
    integer, 33
    long integer, 33
    numeric, 33
    octal, 33
LittleEndianStructure (class in ctypes), 618
ljust() (in module string), 88
ljust() (str method), 42
LK_LOCK (in module msvcrt), 1371
LK_NBLCK (in module msvcrt), 1371
LK_NBRLCK (in module msvcrt), 1371
LK_RLCK (in module msvcrt), 1371
LK_UNLCK (in module msvcrt), 1371
LMTP (class in smtplib), 973
ln() (decimal.Context method), 249
ln() (decimal.Decimal method), 242
LNAME, 554
lngettext() (gettext.GNUTranslations method), 1054
lngettext() (gettext.NullTranslations method), 1052
lngettext() (in module gettext), 1050
load() (Cookie.BaseCookie method), 1012
load() (cookielib.FileCookieJar method), 1005
load() (in module hotshot.stats), 1234
load() (in module json), 800
load() (in module marshal), 332
load() (in module pickle), 319
load() (pickle.Unpickler method), 321
LOAD_ATTR (opcode), 1348
load_cert_chain() (ssl.SSLContext method), 740
LOAD_CLOSURE (opcode), 1349
load_compiled() (in module imp), 1307
LOAD_CONST (opcode), 1347
load_default_certs() (ssl.SSLContext method), 740
LOAD_DEREF (opcode), 1349
load_dh_params() (ssl.SSLContext method), 742
load_dynamic() (in module imp), 1307
load_extension() (sqlite3.Connection method), 349
LOAD_FAST (opcode), 1348
LOAD_GLOBAL (opcode), 1348
load_global() (pickle protocol), 325
LOAD_LOCALS (opcode), 1346
load_module() (in module imp), 1306
load_module() (zipimport.zipimporter method), 1313
LOAD_NAME (opcode), 1347
load_source() (in module imp), 1307
load_verify_locations() (ssl.SSLContext method), 740
loader, 1456
LoadError, 1002
LoadKey() (in module _winreg), 1375
LoadLibrary() (ctypes.LibraryLoader method), 607
loads() (in module json), 800
loads() (in module marshal), 333
loads() (in module pickle), 319
loads() (in module xmlrpclib), 1022
loadTestsFromModule() (unittest.TestLoader method), 1195
loadTestsFromName() (unittest.TestLoader method), 1195

loadTestsFromNames() (unittest.TestLoader method), 1195
loadTestsFromTestCase() (unittest.TestLoader method), 1195
local (class in threading), 627
localcontext() (in module decimal), 246
LOCALE (in module re), 94
locale (module), 1059
localeconv() (in module locale), 1059
LocaleHTMLCalendar (class in calendar), 172
LocaleTextCalendar (class in calendar), 172
localName (xml.dom.Attr attribute), 879
localName (xml.dom.Node attribute), 875
locals() (built-in function), 14
localtime() (in module time), 458
Locator (class in xml.sax.xmlreader), 896
Lock (class in multiprocessing), 652
Lock() (in module threading), 627
lock() (mailbox.Babyl method), 813
lock() (mailbox.Mailbox method), 809
lock() (mailbox.Maildir method), 810
lock() (mailbox.mbox method), 811
lock() (mailbox.MH method), 812
lock() (mailbox.MMDF method), 814
Lock() (multiprocessing.managers.SyncManager method), 659
lock() (mutex.mutex method), 204
lock() (posixfile.posixfile method), 1394
lock_held() (in module imp), 1306
locked() (thread.lock method), 638
lockf() (in module fcntl), 1391
locking() (in module msvcrt), 1371
LockType (in module thread), 637
log() (in module cmath), 231
log() (in module logging), 532
log() (in module math), 228
log() (logging.Logger method), 524
log10() (decimal.Context method), 249
log10() (decimal.Decimal method), 242
log10() (in module cmath), 232
log10() (in module math), 228
log1p() (in module math), 228
log_date_time_string() (BaseHTTPServer.BaseHTTPRequestHandler method), 999
log_error() (BaseHTTPServer.BaseHTTPRequestHandler method), 999
log_exception() (wsgiref.handlers.BaseHandler method), 929
log_message() (BaseHTTPServer.BaseHTTPRequestHandler method), 999
log_request() (BaseHTTPServer.BaseHTTPRequestHandler method), 998
log_to_stderr() (in module multiprocessing), 668
logb() (decimal.Context method), 249
logb() (decimal.Decimal method), 242
Logger (class in logging), 522
LoggerAdapter (class in logging), 531
logging
    Errors, 521
logging (module), 521
logging.config (module), 535
logging.handlers (module), 545
Logical (class in aetypes), 1424
logical_and() (decimal.Context method), 249
logical_and() (decimal.Decimal method), 242
logical_invert() (decimal.Context method), 249
logical_invert() (decimal.Decimal method), 242
logical_or() (decimal.Context method), 249
logical_or() (decimal.Decimal method), 242
logical_xor() (decimal.Context method), 249
logical_xor() (decimal.Decimal method), 242
login() (ftplib.FTP method), 958
login() (imaplib.IMAP4 method), 965
login() (smtplib.SMTP method), 975
login_cram_md5() (imaplib.IMAP4 method), 965
LOGNAME, 417, 554
lognormvariate() (in module random), 265
logout() (imaplib.IMAP4 method), 965
LogRecord (class in logging), 528
long
    built-in function, 33, 86
    integer division, 34
    integer literals, 33
long (2to3 fixer), 1206
long (built-in class), 14
long integer
    object, 33
long_info (in module sys), 1254
longMessage (unittest.TestCase attribute), 1192
longname() (in module curses), 557
LongType (in module types), 214
lookup() (in module codecs), 125
lookup() (in module unicodedata), 139
lookup() (symtable.SymbolTable method), 1331
lookup() (ttk.Style method), 1102
lookup_error() (in module codecs), 126
LookupError, 68
loop() (in module asyncore), 756
lower() (in module string), 86
lower() (str method), 42
lowercase (in module string), 76
LPAR (in module token), 1333
lseek() (in module os), 423
lshift() (in module operator), 284
LSQB (in module token), 1333
lstat() (in module os), 428
lstrip() (in module string), 87

lstrip() (str method), 42
lsub() (imaplib.IMAP4 method), 965
lt() (in module operator), 282
lt() (in module turtle), 1114
Lundh, Fredrik, 1442
LWPCookieJar (class in cookielib), 1006

M

M (in module re), 94
mac_ver() (in module platform), 579
macerrors
    module, 1405
macerrors (module), 1448
machine() (in module platform), 577
MacOS (module), 1404
macostools (module), 1406
macpath (module), 315
macresource (module), 1448
macros (netrc.netrc attribute), 403
mailbox
    module, 836
Mailbox (class in mailbox), 806
mailbox (module), 806
mailcap (module), 805
Maildir (class in mailbox), 809
MaildirMessage (class in mailbox), 815
MailmanProxy (class in smtpd), 978
main() (in module py_compile), 1339
main() (in module unittest), 1199
mainloop() (FrameWork.Application method), 1411
mainloop() (in module turtle), 1128
major() (in module os), 429
make_archive() (in module shutil), 313
MAKE_CLOSURE (opcode), 1349
make_cookies() (cookielib.CookieJar method), 1004
make_form() (in module fl), 1433
MAKE_FUNCTION (opcode), 1349
make_header() (in module email.header), 781
make_msgid() (in module email.utils), 786
make_parser() (in module xml.sax), 888
make_server() (in module wsgiref.simple_server), 925
makedev() (in module os), 429
makedirs() (in module os), 429
makeelement() (xml.etree.ElementTree.Element method), 868
makefile() (socket.socket method), 721
makefolder() (mhlib.MH method), 826
makeLogRecord() (in module logging), 533
makePickle() (logging.handlers.SocketHandler method), 548
makeRecord() (logging.Logger method), 525
makeSocket() (logging.handlers.DatagramHandler method), 549
makeSocket() (logging.handlers.SocketHandler method), 548
maketrans() (in module string), 85
makeusermenus() (FrameWork.Application method), 1411
map (2to3 fixer), 1206
map() (built-in function), 14
map() (in module future_builtins), 1264
map() (multiprocessing.pool.multiprocessing.Pool method), 664
map() (ttk.Style method), 1101
map_async() (multiprocessing.pool.multiprocessing.Pool method), 664
map_table_b2() (in module stringprep), 142
map_table_b3() (in module stringprep), 142
mapcolor() (in module fl), 1434
mapfile() (ic.IC method), 1404
mapfile() (in module ic), 1403
mapLogRecord() (logging.handlers.HTTPHandler method), 553
mapping, 1456
mapping
    object, 53
    types, operations on, 53
Mapping (class in collections), 188
mapping() (msilib.Control method), 1369
MappingView (class in collections), 188
mapPriority() (logging.handlers.SysLogHandler method), 550
maps() (in module nis), 1399
maptypecreator() (ic.IC method), 1404
maptypecreator() (in module ic), 1403
marshal (module), 331
marshalling
    objects, 317
masking
    operations, 35
match() (in module nis), 1399
match() (in module re), 95
match() (re.RegexObject method), 97
match_hostname() (in module ssl), 730
MatchObject (class in re), 98
math
    module, 34, 233
math (module), 226
max
    built-in function, 38
max (datetime.date attribute), 150
max (datetime.datetime attribute), 154
max (datetime.time attribute), 159
max (datetime.timedelta attribute), 147
max() (built-in function), 14
max() (decimal.Context method), 250
max() (decimal.Decimal method), 242

max() (in module audioop), 1031
MAX_INTERPOLATION_DEPTH (in module ConfigParser), 397
max_mag() (decimal.Context method), 250
max_mag() (decimal.Decimal method), 242
maxarray (repr.Repr attribute), 221
maxdeque (repr.Repr attribute), 221
maxdict (repr.Repr attribute), 221
maxDiff (unittest.TestCase attribute), 1192
maxfrozenset (repr.Repr attribute), 221
maxint (in module sys), 1254
maxlen (collections.deque attribute), 178
MAXLEN (in module mimify), 833
maxlevel (repr.Repr attribute), 221
maxlist (repr.Repr attribute), 221
maxlong (repr.Repr attribute), 221
maxother (repr.Repr attribute), 221
maxpp() (in module audioop), 1031
maxset (repr.Repr attribute), 221
maxsize (in module sys), 1254
maxstring (repr.Repr attribute), 221
maxtuple (repr.Repr attribute), 221
maxunicode (in module sys), 1254
maxval (EasyDialogs.ProgressBar attribute), 1409
MAXYEAR (in module datetime), 145
MB_ICONASTERISK (in module winsound), 1382
MB_ICONEXCLAMATION (in module winsound), 1382
MB_ICONHAND (in module winsound), 1382
MB_ICONQUESTION (in module winsound), 1382
MB_OK (in module winsound), 1382
mbox (class in mailbox), 811
mboxMessage (class in mailbox), 817
md5 (module), 413
md5() (in module md5), 413
MemberDescriptorType (in module types), 215
memmove() (in module ctypes), 613
MemoryError, 69
MemoryHandler (class in logging.handlers), 552
memoryview (built-in class), 61
memset() (in module ctypes), 613
Menu() (in module FrameWork), 1410
MenuBar() (in module FrameWork), 1410
MenuItem() (in module FrameWork), 1410
merge() (in module heapq), 190
Message (class in email.message), 766
Message (class in mailbox), 814
Message (class in mhlib), 826
Message (class in mimetools), 828
Message (class in rfc822), 836
Message (in module mimetools), 998
message digest, MD5, 409, 413
Message() (in module EasyDialogs), 1408
message_from_file() (in module email), 774
message_from_string() (in module email), 774
MessageBeep() (in module winsound), 1380
MessageClass (BaseHTTPServer.BaseHTTPRequestHandler attribute), 998
MessageError, 784
MessageParseError, 784
meta() (in module curses), 557
meta_path (in module sys), 1254
metaclass, 1456
metaclass (2to3 fixer), 1206
metavar (optparse.Option attribute), 506
Meter (class in Tix), 1106
method, 1456
method
  object, 64
method resolution order, 1456
methodattrs (2to3 fixer), 1206
methodcaller() (in module operator), 289
methodHelp() (xmlrpclib.ServerProxy.system method), 1017
methods
  string, 39
methods (pyclbr.Class attribute), 1338
methodSignature() (xmlrpclib.ServerProxy.system method), 1017
MethodType (in module types), 214
MH (class in mailbox), 812
MH (class in mhlib), 826
mhlib (module), 826
MHMailbox (class in mailbox), 824
MHMessage (class in mailbox), 818
microsecond (datetime.datetime attribute), 154
microsecond (datetime.time attribute), 160
MIME
  base64 encoding, 840
  content type, 829
  headers, 829, 914
  quoted-printable encoding, 844
mime_decode_header() (in module mimify), 833
mime_encode_header() (in module mimify), 833
MIMEApplication (class in email.mime.application), 777
MIMEAudio (class in email.mime.audio), 778
MIMEBase (class in email.mime.base), 777
MIMEImage (class in email.mime.image), 778
MIMEMessage (class in email.mime.message), 778
MIMEMultipart (class in email.mime.multipart), 777
MIMENonMultipart (class in email.mime.nonmultipart), 777
MIMEText (class in email.mime.text), 778
mimetools
  module, 931
mimetools (module), 828
MimeTypes (class in mimetypes), 831

mimetypes (module), 829
MimeWriter (class in MimeWriter), 832
MimeWriter (module), 832
mimify (module), 833
mimify() (in module mimify), 833
min
    built-in function, 38
min (datetime.date attribute), 149
min (datetime.datetime attribute), 154
min (datetime.time attribute), 159
min (datetime.timedelta attribute), 147
min() (built-in function), 14
min() (decimal.Context method), 250
min() (decimal.Decimal method), 243
min_mag() (decimal.Context method), 250
min_mag() (decimal.Decimal method), 243
MINEQUAL (in module token), 1333
MiniAEFrame (module), 1424
MiniApplication (class in MiniAEFrame), 1424
minmax() (in module audioop), 1031
minor() (in module os), 429
MINUS (in module token), 1333
minus() (decimal.Context method), 250
minute (datetime.datetime attribute), 154
minute (datetime.time attribute), 160
MINYEAR (in module datetime), 145
mirrored() (in module unicodedata), 140
misc_header (cmd.Cmd attribute), 1069
MissingSectionHeaderError, 397
MIXERDEV, 1044
mkalias() (in module macostools), 1406
mkd() (ftplib.FTP method), 959
mkdir() (in module os), 429
mkdtemp() (in module tempfile), 305
mkfifo() (in module os), 428
mknod() (in module os), 429
mkstemp() (in module tempfile), 305
mktemp() (in module tempfile), 306
mktime() (in module time), 458
mktime_tz() (in module email.utils), 786
mktime_tz() (in module rfc822), 837
mmap (class in mmap), 693
mmap (module), 692
MMDF (class in mailbox), 814
MmdfMailbox (class in mailbox), 824
MMDFMessage (class in mailbox), 821
mod() (in module operator), 284
mode (file attribute), 60
mode (io.FileIO attribute), 451
mode (ossaudiodev.oss_audio_device attribute), 1047
mode (tarfile.TarInfo attribute), 382
mode() (in module turtle), 1133
modf() (in module math), 227
modified() (robotparser.RobotFileParser method), 402
Modify() (msilib.View method), 1367
modify() (select.epoll method), 623
modify() (select.poll method), 624
module, 1457
module
    __main__, 1318, 1319
    _locale, 1059
    AL, 1427
    base64, 842
    bdb, 1219
    binhex, 842
    bsddb, 330, 333, 337
    CGIHTTPServer, 996
    cmd, 1219
    copy, 328
    cPickle, 328
    crypt, 1384
    dbhash, 333
    dbm, 330, 333, 336
    dumbdbm, 333
    errno, 69, 714
    fcntl, 58
    formatter, 855
    FrameWork, 1424
    gdbm, 330, 333
    glob, 308
    htmllib, 936
    icglue, 1403
    imp, 24
    knee, 1308, 1312
    macerrors, 1405
    mailbox, 836
    math, 34, 233
    mimetools, 931
    os, 57, 1383
    pickle, 217, 328, 329, 332
    pty, 424
    pwd, 292
    pyexpat, 900
    re, 48, 75, 308
    rfc822, 828
    search path, 309, 1255, 1290
    sgmllib, 855
    shelve, 332
    signal, 638
    SimpleHTTPServer, 996
    sitecustomize, 1290
    socket, 57, 911
    stat, 432
    statvfs, 432
    string, 48, 1064, 1065
    struct, 723

SUNAUDIODEV, 1445
sunaudiodev, 1446
types, 65
urllib, 949
urlparse, 937
usercustomize, 1290
uu, 842
module (pyclbr.Class attribute), 1338
module (pyclbr.Function attribute), 1338
module() (in module new), 216
ModuleFinder (class in modulefinder), 1317
modulefinder (module), 1316
modules (in module sys), 1255
modules (modulefinder.ModuleFinder attribute), 1317
ModuleType (in module types), 215
mono2grey() (in module imageop), 1033
month (datetime.date attribute), 150
month (datetime.datetime attribute), 154
month() (in module calendar), 173
month_abbr (in module calendar), 174
month_name (in module calendar), 173
monthcalendar() (in module calendar), 173
monthdatescalendar() (calendar.Calendar method), 171
monthdays2calendar() (calendar.Calendar method), 171
monthdayscalendar() (calendar.Calendar method), 171
monthrange() (in module calendar), 173
Morsel (class in Cookie), 1013
most_common() (collections.Counter method), 175
mouseinterval() (in module curses), 557
mousemask() (in module curses), 557
move() (curses.panel.Panel method), 576
move() (curses.window method), 565
move() (in module findertools), 1407
move() (in module shutil), 311
move() (mmap.mmap method), 695
move() (ttk.Treeview method), 1099
movemessage() (mhlib.Folder method), 827
MozillaCookieJar (class in cookielib), 1006
MRO, 1457
mro() (class method), 66
msftoframe() (in module cd), 1429
msg (httplib.HTTPResponse attribute), 954
msg() (telnetlib.Telnet method), 979
msi, 1365
msilib (module), 1365
msvcrt (module), 1370
mt_interact() (telnetlib.Telnet method), 980
mtime (tarfile.TarInfo attribute), 382
mtime() (robotparser.RobotFileParser method), 402
mul() (in module audioop), 1031
mul() (in module operator), 284
MultiCall (class in xmlrpclib), 1021
MultiFile (class in multifile), 834
multifile (module), 834
MULTILINE (in module re), 94
MultipartConversionError, 785
multiply() (decimal.Context method), 250
multiprocessing (module), 639
multiprocessing.connection (module), 665
multiprocessing.dummy (module), 669
multiprocessing.Manager() (in module multiprocessing.sharedctypes), 657
multiprocessing.managers (module), 657
multiprocessing.Pool (class in multiprocessing.pool), 663
multiprocessing.pool (module), 663
multiprocessing.queues.SimpleQueue (class in multiprocessing), 649
multiprocessing.sharedctypes (module), 655
mutable, 1457
mutable
sequence types, 48
MutableMapping (class in collections), 188
MutableSequence (class in collections), 188
MutableSet (class in collections), 188
MutableString (class in UserString), 213
mutex (class in mutex), 203
mutex (module), 203
mvderwin() (curses.window method), 565
mvwin() (curses.window method), 565
myrights() (imaplib.IMAP4 method), 965

N

N_TOKENS (in module token), 1333
name (cookielib.Cookie attribute), 1009
name (doctest.DocTest attribute), 1169
name (file attribute), 60
name (in module os), 415
NAME (in module token), 1333
name (io.FileIO attribute), 451
name (multiprocessing.Process attribute), 645
name (ossaudiodev.oss_audio_device attribute), 1047
name (pyclbr.Class attribute), 1338
name (pyclbr.Function attribute), 1338
name (tarfile.TarInfo attribute), 382
name (threading.Thread attribute), 630
name (xml.dom.Attr attribute), 879
name (xml.dom.DocumentType attribute), 877
name() (in module unicodedata), 139
name2codepoint (in module htmlentitydefs), 857
named tuple, 1457
NamedTemporaryFile() (in module tempfile), 304
namedtuple() (in module collections), 181

NameError, 69
namelist() (zipfile.ZipFile method), 372
nameprep() (in module encodings.idna), 139
namespace, 1457
Namespace (class in argparse), 484
Namespace (class in multiprocessing.managers), 659
namespace() (imaplib.IMAP4 method), 965
Namespace() (multiprocessing.managers.SyncManager method), 659
NAMESPACE_DNS (in module uuid), 982
NAMESPACE_OID (in module uuid), 982
NAMESPACE_URL (in module uuid), 982
NAMESPACE_X500 (in module uuid), 983
NamespaceErr, 881
namespaceURI (xml.dom.Node attribute), 875
NaN, 11, 86
NannyNag, 1337
napms() (in module curses), 558
nargs (optparse.Option attribute), 506
Nav (module), 1448
Navigation Services, 1409
ndiff() (in module difflib), 112
ndim (memoryview attribute), 62
ne (2to3 fixer), 1206
ne() (in module operator), 282
neg() (in module operator), 284
nested scope, 1457
nested() (in module contextlib), 1270
netrc (class in netrc), 403
netrc (module), 403
NetrcParseError, 403
netscape (cookielib.CookiePolicy attribute), 1007
Network News Transfer Protocol, 968
new (module), 216
new() (in module hmac), 412
new() (in module md5), 413
new() (in module sha), 414
new-style class, 1457
new_alignment() (formatter.writer method), 1361
new_font() (formatter.writer method), 1361
new_margin() (formatter.writer method), 1361
new_module() (in module imp), 1306
new_panel() (in module curses.panel), 575
new_spacing() (formatter.writer method), 1362
new_styles() (formatter.writer method), 1362
newconfig() (in module al), 1427
newgroups() (nntplib.NNTP method), 970
NEWLINE (in module token), 1333
newlines (file attribute), 60
newlines (io.TextIOBase attribute), 453
newnews() (nntplib.NNTP method), 970
newpad() (in module curses), 558
newwin() (in module curses), 558
next (2to3 fixer), 1206
next() (bsddb.bsddbobject method), 340
next() (built-in function), 15
next() (csv.csvreader method), 392
next() (dbhash.dbhash method), 338
next() (file method), 58
next() (iterator method), 37
next() (mailbox.oldmailbox method), 823
next() (multifile.MultiFile method), 835
next() (nntplib.NNTP method), 971
next() (tarfile.TarFile method), 380
next() (ttk.Treeview method), 1100
next_minus() (decimal.Context method), 250
next_minus() (decimal.Decimal method), 243
next_plus() (decimal.Context method), 250
next_plus() (decimal.Decimal method), 243
next_toward() (decimal.Context method), 250
next_toward() (decimal.Decimal method), 243
nextfile() (in module fileinput), 296
nextkey() (in module gdbm), 336
nextpart() (MimeWriter.MimeWriter method), 832
nextSibling (xml.dom.Node attribute), 875
ngettext() (gettext.GNUTranslations method), 1054
ngettext() (gettext.NullTranslations method), 1052
ngettext() (in module gettext), 1050
nice() (in module os), 439
nis (module), 1399
NIST, 414
NL (in module tokenize), 1335
nl() (in module curses), 558
nl_langinfo() (in module locale), 1060
nlargest() (in module heapq), 190
nlst() (ftplib.FTP method), 959
NNTP
    protocol, 968
NNTP (class in nntplib), 969
NNTPDataError, 970
NNTPError, 969
nntplib (module), 968
NNTPPermanentError, 969
NNTPProtocolError, 970
NNTPReplyError, 969
NNTPTemporaryError, 969
no_proxy, 932
nocbreak() (in module curses), 558
NoDataAllowedErr, 881
Node (class in compiler.ast), 1352
node() (in module platform), 577
nodelay() (curses.window method), 565
nodeName (xml.dom.Node attribute), 875
NodeTransformer (class in ast), 1330
nodeType (xml.dom.Node attribute), 875
nodeValue (xml.dom.Node attribute), 875
NodeVisitor (class in ast), 1329
NODISC (in module cd), 1430

noecho() (in module curses), 558
NOEXPR (in module locale), 1061
nofill (htmllib.HTMLParser attribute), 856
nok_builtin_names (rexec.RExec attribute), 1302
noload() (pickle.Unpickler method), 321
NoModificationAllowedErr, 881
nonblock() (ossaudiodev.oss_audio_device method), 1045
None (Built-in object), 31
None (built-in variable), 29
NoneType (in module types), 214
nonl() (in module curses), 558
nonzero (2to3 fixer), 1206
noop() (imaplib.IMAP4 method), 966
noop() (poplib.POP3 method), 962
NoOptionError, 397
NOP (opcode), 1343
noqiflush() (in module curses), 558
noraw() (in module curses), 558
normalize() (decimal.Context method), 250
normalize() (decimal.Decimal method), 243
normalize() (in module locale), 1063
normalize() (in module unicodedata), 140
normalize() (xml.dom.Node method), 876
NORMALIZE_WHITESPACE (in module doctest), 1160
normalvariate() (in module random), 265
normcase() (in module os.path), 293
normpath() (in module os.path), 293
NoSectionError, 397
NoSuchMailboxError, 822
not
    operator, 32
not in
    operator, 33, 38
not_() (in module operator), 283
NotANumber, 143
notationDecl() (xml.sax.handler.DTDHandler method), 894
NotationDeclHandler() (xml.parsers.expat.xmlparser method), 905
notations (xml.dom.DocumentType attribute), 877
NotConnected, 950
NoteBook (class in Tix), 1107
Notebook (class in ttk), 1093
NotEmptyError, 822
NOTEQUAL (in module token), 1333
NotFoundErr, 881
notify() (threading.Condition method), 633
notify_all() (threading.Condition method), 634
notifyAll() (threading.Condition method), 634
notimeout() (curses.window method), 565
NotImplemented (built-in variable), 29
NotImplementedError, 69
NotImplementedType (in module types), 215
NotStandaloneHandler() (xml.parsers.expat.xmlparser method), 905
NotSupportedErr, 881
noutrefresh() (curses.window method), 565
now() (datetime.datetime class method), 153
NProperty (class in aetypes), 1424
NSIG (in module signal), 752
nsmallest() (in module heapq), 190
NT_OFFSET (in module token), 1333
NTEventLogHandler (class in logging.handlers), 551
ntohl() (in module socket), 718
ntohs() (in module socket), 718
ntransfercmd() (ftplib.FTP method), 959
NullFormatter (class in formatter), 1361
NullHandler (class in logging), 546
NullImporter (class in imp), 1308
NullTranslations (class in gettext), 1052
NullWriter (class in formatter), 1362
Number (class in numbers), 223
NUMBER (in module token), 1333
number_class() (decimal.Context method), 250
number_class() (decimal.Decimal method), 243
numbers (module), 223
numerator (numbers.Rational attribute), 224
numeric
    conversions, 34
    literals, 33
    object, 32, 33
    types, operations on, 34
numeric() (in module unicodedata), 140
Numerical Python, 20
numliterals (2to3 fixer), 1206
nurbscurve() (in module gl), 1440
nurbssurface() (in module gl), 1440
nvarray() (in module gl), 1440

O

O_APPEND (in module os), 425
O_ASYNC (in module os), 425
O_BINARY (in module os), 425
O_CREAT (in module os), 425
O_DIRECT (in module os), 425
O_DIRECTORY (in module os), 425
O_DSYNC (in module os), 425
O_EXCL (in module os), 425
O_EXLOCK (in module os), 425
O_NDELAY (in module os), 425
O_NOATIME (in module os), 425
O_NOCTTY (in module os), 425
O_NOFOLLOW (in module os), 425
O_NOINHERIT (in module os), 425
O_NONBLOCK (in module os), 425

O_RANDOM (in module os), 425
O_RDONLY (in module os), 425
O_RDWR (in module os), 425
O_RSYNC (in module os), 425
O_SEQUENTIAL (in module os), 425
O_SHLOCK (in module os), 425
O_SHORT_LIVED (in module os), 425
O_SYNC (in module os), 425
O_TEMPORARY (in module os), 425
O_TEXT (in module os), 425
O_TRUNC (in module os), 425
O_WRONLY (in module os), 425
object, 1457
object
    Boolean, 33
    buffer, 37
    bytearray, 37
    code, 64, 332
    complex number, 33
    dictionary, 53
    file, 57
    floating point, 33
    integer, 33
    list, 37, 48
    long integer, 33
    mapping, 53
    method, 64
    numeric, 32, 33
    sequence, 37
    set, 50
    socket, 713
    string, 37
    traceback, 1248, 1276
    tuple, 37
    type, 23
    Unicode, 37
    xrange, 37, 48
object (built-in class), 15
object (exceptions.UnicodeError attribute), 71
objects
    comparing, 32
    flattening, 317
    marshalling, 317
    persistent, 317
    pickling, 317
    serializing, 317
ObjectSpecifier (class in aetypes), 1424
obufcount() (ossaudiodev.oss_audio_device method), 1046
obuffree() (ossaudiodev.oss_audio_device method), 1046
oct() (built-in function), 15
oct() (in module future_builtins), 1264
octal
    literals, 33
octdigits (in module string), 76
offset (xml.parsers.expat.ExpatError attribute), 906
OK (in module curses), 567
ok_builtin_modules (rexec.RExec attribute), 1302
ok_file_types (rexec.RExec attribute), 1302
ok_path (rexec.RExec attribute), 1302
ok_posix_names (rexec.RExec attribute), 1302
ok_sys_names (rexec.RExec attribute), 1302
OleDLL (class in ctypes), 606
onclick() (in module turtle), 1127, 1132
ondrag() (in module turtle), 1127
onecmd() (cmd.Cmd method), 1068
onkey() (in module turtle), 1132
onrelease() (in module turtle), 1127
onscreenclick() (in module turtle), 1132
ontimer() (in module turtle), 1133
OP (in module token), 1333
OP_ALL (in module ssl), 734
OP_CIPHER_SERVER_PREFERENCE (in module ssl), 735
OP_ENABLE_MIDDLEBOX_COMPAT (in module ssl), 735
OP_NO_COMPRESSION (in module ssl), 735
OP_NO_SSLv2 (in module ssl), 734
OP_NO_SSLv3 (in module ssl), 734
OP_NO_TLSv1 (in module ssl), 734
OP_NO_TLSv1_1 (in module ssl), 734
OP_NO_TLSv1_2 (in module ssl), 734
OP_NO_TLSv1_3 (in module ssl), 735
OP_SINGLE_DH_USE (in module ssl), 735
OP_SINGLE_ECDH_USE (in module ssl), 735
Open Scripting Architecture, 1424
open() (built-in function), 15
open() (FrameWork.DialogWindow method), 1413
open() (FrameWork.Window method), 1412
open() (imaplib.IMAP4 method), 966
open() (in module aifc), 1033
open() (in module anydbm), 333
open() (in module cd), 1430
open() (in module codecs), 127
open() (in module dbhash), 337
open() (in module dbm), 335
open() (in module dl), 1387
open() (in module dumbdbm), 341
open() (in module gdbm), 336
open() (in module gzip), 367
open() (in module io), 445
open() (in module os), 423
open() (in module ossaudiodev), 1044
open() (in module posixfile), 1394
open() (in module shelve), 329
open() (in module sunau), 1036
open() (in module sunaudiodev), 1445

open() (in module tarfile), 377
open() (in module wave), 1038
open() (in module webbrowser), 911
open() (pipes.Template method), 1394
open() (tarfile.TarFile class method), 380
open() (telnetlib.Telnet method), 979
open() (urllib.URLopener method), 935
open() (urllib2.OpenerDirector method), 942
open() (webbrowser.controller method), 913
open() (zipfile.ZipFile method), 372
open_new() (in module webbrowser), 912
open_new() (webbrowser.controller method), 914
open_new_tab() (in module webbrowser), 912
open_new_tab() (webbrowser.controller method), 914
open_osfhandle() (in module msvcrt), 1371
open_unknown() (urllib.URLopener method), 935
OpenDatabase() (in module msilib), 1365
opendir() (in module dircache), 314
OpenerDirector (class in urllib2), 939
openfolder() (mhlib.MH method), 826
openfp() (in module sunau), 1036
openfp() (in module wave), 1038
OpenGL, 1441
OpenKey() (in module _winreg), 1375
OpenKeyEx() (in module _winreg), 1375
openlog() (in module syslog), 1400
openmessage() (mhlib.Message method), 827
openmixer() (in module ossaudiodev), 1044
openport() (in module al), 1427
openpty() (in module os), 424
openpty() (in module pty), 1390
openrf() (in module MacOS), 1405
OpenSSL
    (use in module hashlib), 409
    (use in module ssl), 726
OPENSSL_VERSION (in module ssl), 736
OPENSSL_VERSION_INFO (in module ssl), 736
OPENSSL_VERSION_NUMBER (in module ssl), 736
OpenView() (msilib.Database method), 1366
operation
    concatenation, 38
    extended slice, 38
    repetition, 38
    slice, 38
    subscript, 38
operations
    bitwise, 35
    Boolean, 31, 32
    masking, 35
    shifting, 35
operations on
    dictionary type, 53
    integer types, 35
    list type, 48
    mapping types, 53
    numeric types, 34
    sequence types, 38, 48
operator
    !=, 32
    *, 33
    **, 33
    +, 33
    -, 33
    /, 33
    //, 33
    ==, 32
    %, 33
    &, 35
    ^, 35
    ~, 35
    |, 35
    >, 32
    >=, 32
    >>, 35
    <, 32
    <=, 32
    <<, 35
    and, 31, 32
    comparison, 32
    in, 33, 38
    is, 32
    is not, 32
    not, 32
    not in, 33, 38
    or, 31, 32
operator (module), 282
opmap (in module dis), 1342
opname (in module dis), 1342
optimize() (in module pickletools), 1350
OptionGroup (class in optparse), 499
OptionMenu (class in Tix), 1106
OptionParser (class in optparse), 503
options (doctest.Example attribute), 1170
options (ssl.SSLContext attribute), 743
options() (ConfigParser.RawConfigParser method), 398
optionxform() (ConfigParser.RawConfigParser method), 399
optparse (module), 492
or
    operator, 31, 32
or_() (in module operator), 284
ord() (built-in function), 16
ordered_attributes (xml.parsers.expat.xmlparser attribute), 903
OrderedDict (class in collections), 185

Ordinal (class in aetypes), 1424
origin_server (wsgiref.handlers.BaseHandler attribute), 930
os
    module, 57, 1383
os (module), 415
os.path (module), 291
os_environ (wsgiref.handlers.BaseHandler attribute), 928
OSError, 69
ossaudiodev (module), 1044
OSSAudioError, 1044
output (subprocess.CalledProcessError attribute), 703
output() (Cookie.BaseCookie method), 1012
output() (Cookie.Morsel method), 1013
output_charset (email.charset.Charset attribute), 781
output_charset() (gettext.NullTranslations method), 1053
output_codec (email.charset.Charset attribute), 782
output_difference() (doctest.OutputChecker method), 1173
OutputChecker (class in doctest), 1172
OutputString() (Cookie.Morsel method), 1014
OutputType (in module cStringIO), 121
Overflow (class in decimal), 252
OverflowError, 69
overlay() (curses.window method), 565
Overmars, Mark, 1433
overwrite() (curses.window method), 565

P

P_DETACH (in module os), 440
P_NOWAIT (in module os), 440
P_NOWAITO (in module os), 440
P_OVERLAY (in module os), 440
P_WAIT (in module os), 440
pack() (in module aepack), 1422
pack() (in module struct), 105
pack() (mailbox.MH method), 812
pack() (struct.Struct method), 109
pack_array() (xdrlib.Packer method), 405
pack_bytes() (xdrlib.Packer method), 405
pack_double() (xdrlib.Packer method), 404
pack_farray() (xdrlib.Packer method), 405
pack_float() (xdrlib.Packer method), 404
pack_fopaque() (xdrlib.Packer method), 404
pack_fstring() (xdrlib.Packer method), 404
pack_into() (in module struct), 105
pack_into() (struct.Struct method), 109
pack_list() (xdrlib.Packer method), 405
pack_opaque() (xdrlib.Packer method), 405
pack_string() (xdrlib.Packer method), 404
package, 1457
package, 1290
Packer (class in xdrlib), 404
packevent() (in module aetools), 1421
packing
    binary data, 104
packing (widgets), 1081
PAGER, 1151, 1222
pair_content() (in module curses), 558
pair_number() (in module curses), 558
PanedWindow (class in Tix), 1107
parameter, 1457
pardir (in module os), 444
paren (2to3 fixer), 1206
parent (urllib2.BaseHandler attribute), 943
parent() (ttk.Treeview method), 1100
parentNode (xml.dom.Node attribute), 875
paretovariate() (in module random), 265
parse() (doctest.DocTestParser method), 1171
parse() (email.parser.Parser method), 774
parse() (in module ast), 1329
parse() (in module cgi), 917
parse() (in module compiler), 1351
parse() (in module xml.dom.minidom), 883
parse() (in module xml.dom.pulldom), 888
parse() (in module xml.etree.ElementTree), 866
parse() (in module xml.sax), 889
parse() (robotparser.RobotFileParser method), 402
parse() (string.Formatter method), 76
parse() (xml.etree.ElementTree.ElementTree method), 869
Parse() (xml.parsers.expat.xmlparser method), 901
parse() (xml.sax.xmlreader.XMLReader method), 897
parse_and_bind() (in module readline), 696
parse_args() (argparse.ArgumentParser method), 481
PARSE_COLNAMES (in module sqlite3), 344
parse_config_h() (in module sysconfig), 1262
PARSE_DECLTYPES (in module sqlite3), 344
parse_header() (in module cgi), 918
parse_known_args() (argparse.ArgumentParser method), 491
parse_multipart() (in module cgi), 918
parse_qs() (in module cgi), 918
parse_qs() (in module urlparse), 985
parse_qsl() (in module cgi), 918
parse_qsl() (in module urlparse), 985
parseaddr() (in module email.utils), 785
parseaddr() (in module rfc822), 837
parsedate() (in module email.utils), 786
parsedate() (in module rfc822), 837
parsedate_tz() (in module email.utils), 786
parsedate_tz() (in module rfc822), 837

parseFile() (in module compiler), 1351
ParseFile() (xml.parsers.expat.xmlparser method), 901
ParseFlags() (in module imaplib), 963
Parser (class in email.parser), 774
parser (module), 1321
ParserCreate() (in module xml.parsers.expat), 900
ParserError, 1324
ParseResult (class in urlparse), 988
parsesequence() (mhlib.Folder method), 827
parsestr() (email.parser.Parser method), 774
parseString() (in module xml.dom.minidom), 883
parseString() (in module xml.dom.pulldom), 888
parseString() (in module xml.sax), 889
parseurl() (ic.IC method), 1404
parseurl() (in module ic), 1403
parsing
  Python source code, 1321
  URL, 984
ParsingError, 397
partial() (imaplib.IMAP4 method), 966
partial() (in module functools), 280
partition() (str method), 42
pass_() (poplib.POP3 method), 961
Paste, 1144
PATH, 436, 439, 444, 911, 919, 921
path
  configuration file, 1290
  module search, 309, 1255, 1290
  operations, 291
path (BaseHTTPServer.BaseHTTPRequestHandler attribute), 997
path (cookielib.Cookie attribute), 1009
path (in module sys), 1255
Path browser, 1141
path_hooks (in module sys), 1255
path_importer_cache (in module sys), 1255
path_return_ok() (cookielib.CookiePolicy method), 1007
pathconf() (in module os), 430
pathconf_names (in module os), 430
pathname2url() (in module urllib), 934
pathsep (in module os), 444
pattern (re.RegexObject attribute), 98
pause() (in module signal), 752
PAUSED (in module cd), 1430
PAX_FORMAT (in module tarfile), 379
pax_headers (tarfile.TarFile attribute), 382
pax_headers (tarfile.TarInfo attribute), 383
pbkdf2_hmac() (in module hashlib), 411
pd() (in module turtle), 1120
Pdb (class in pdb), 1219, 1221
pdb (module), 1219
peek() (io.BufferedReader method), 452
PEM_cert_to_DER_cert() (in module ssl), 731
pen() (in module turtle), 1120
pencolor() (in module turtle), 1122
PendingDeprecationWarning, 72
pendown() (in module turtle), 1120
pensize() (in module turtle), 1120
penup() (in module turtle), 1120
PEP, 1458
PERCENT (in module token), 1333
PERCENTEQUAL (in module token), 1333
Performance, 1234
permutations() (in module itertools), 273
Persist() (msilib.SummaryInformation method), 1367
persistence, 317
persistent
  objects, 317
persistent_id (pickle protocol), 324
persistent_load (pickle protocol), 324
pformat() (in module pprint), 218
pformat() (pprint.PrettyPrinter method), 219
phase() (in module cmath), 231
pi (in module cmath), 233
pi (in module math), 230
pick() (in module gl), 1440
pickle
  module, 217, 328, 329, 332
pickle (module), 317
pickle() (in module copy_reg), 328
PickleError, 319
Pickler (class in pickle), 320
pickletools (module), 1350
pickling
  objects, 317
PicklingError, 319
pid (multiprocessing.Process attribute), 645
pid (popen2.Popen3 attribute), 755
pid (subprocess.Popen attribute), 708
PIL (the Python Imaging Library), 1442
PIPE (in module subprocess), 703
Pipe() (in module multiprocessing), 647
pipe() (in module os), 424
PIPE_BUF (select.select attribute), 622
pipes (module), 1392
PixMapWrapper (module), 1448
PKG_DIRECTORY (in module imp), 1306
pkgutil (module), 1314
platform (in module sys), 1255
platform (module), 576
platform() (in module platform), 577
PLAYING (in module cd), 1430
PlaySound() (in module winsound), 1380
plist
  file, 407

plistlib (module), 407
plock() (in module os), 439
PLUS (in module token), 1333
plus() (decimal.Context method), 250
PLUSEQUAL (in module token), 1333
pm() (in module pdb), 1221
pnum (in module cd), 1430
POINTER() (in module ctypes), 613
pointer() (in module ctypes), 613
polar() (in module cmath), 231
poll() (Connection method), 651
poll() (in module select), 621
poll() (popen2.Popen3 method), 755
poll() (select.epoll method), 623
poll() (select.poll method), 624
poll() (subprocess.Popen method), 707
pop() (array.array method), 197
pop() (asynchat.fifo method), 762
pop() (collections.deque method), 177
pop() (dict method), 55
pop() (frozenset method), 52
pop() (list method), 48
pop() (mailbox.Mailbox method), 808
pop() (multifile.MultiFile method), 835
POP3
    protocol, 960
POP3 (class in poplib), 960
POP3_SSL (class in poplib), 961
pop_alignment() (formatter.formatter method), 1360
POP_BLOCK (opcode), 1346
pop_font() (formatter.formatter method), 1360
POP_JUMP_IF_FALSE (opcode), 1348
POP_JUMP_IF_TRUE (opcode), 1348
pop_margin() (formatter.formatter method), 1360
pop_source() (shlex.shlex method), 1071
pop_style() (formatter.formatter method), 1361
POP_TOP (opcode), 1343
Popen (class in subprocess), 704
popen() (in module os), 420, 622
popen() (in module platform), 579
popen2 (module), 754
popen2() (in module os), 421
popen2() (in module popen2), 754
Popen3 (class in popen2), 754
popen3() (in module os), 421
popen3() (in module popen2), 754
Popen4 (class in popen2), 754
popen4() (in module os), 421
popen4() (in module popen2), 754
popitem() (collections.OrderedDict method), 185
popitem() (dict method), 55
popitem() (mailbox.Mailbox method), 808
popleft() (collections.deque method), 178
poplib (module), 960
PopupMenu (class in Tix), 1106
port (cookielib.Cookie attribute), 1009
port_specified (cookielib.Cookie attribute), 1010
PortableUnixMailbox (class in mailbox), 823
pos (re.MatchObject attribute), 100
pos() (in module operator), 284
pos() (in module turtle), 1118
position() (in module turtle), 1118
positional argument, 1458
POSIX
    file object, 1394
    I/O control, 1388
    threads, 637
posix (module), 1383
posix (tarfile.TarFile attribute), 382
posixfile (module), 1394
POSIXLY_CORRECT, 520
post() (nntplib.NNTP method), 971
post() (ossaudiodev.oss_audio_device method), 1046
post_mortem() (in module pdb), 1221
postcmd() (cmd.Cmd method), 1068
postloop() (cmd.Cmd method), 1069
pow() (built-in function), 16
pow() (in module math), 228
pow() (in module operator), 284
power() (decimal.Context method), 250
pprint (module), 217
pprint() (bdb.Breakpoint method), 1215
pprint() (in module pprint), 218
pprint() (pprint.PrettyPrinter method), 219
prcal() (in module calendar), 173
preamble (email.message.Message attribute), 772
precmd() (cmd.Cmd method), 1068
prefix (in module sys), 1256
prefix (xml.dom.Attr attribute), 879
prefix (xml.dom.Node attribute), 875
prefix (zipimport.zipimporter attribute), 1314
PREFIXES (in module site), 1290
preloop() (cmd.Cmd method), 1068
preorder() (compiler.visitor.ASTVisitor method), 1358
prepare_input_source() (in module xml.sax.saxutils), 896
prepend() (pipes.Template method), 1394
PrettyPrinter (class in pprint), 218
prev() (ttk.Treeview method), 1100
previous() (bsddb.bsddbobject method), 340
previous() (dbhash.dbhash method), 338
previousSibling (xml.dom.Node attribute), 875
print
    statement, 31
print (2to3 fixer), 1206

print() (built-in function), 16
Print() (in module findertools), 1407
print_callees() (pstats.Stats method), 1230
print_callers() (pstats.Stats method), 1230
print_directory() (in module cgi), 918
print_environ() (in module cgi), 918
print_environ_usage() (in module cgi), 918
print_exc() (in module traceback), 1276
print_exc() (timeit.Timer method), 1236
print_exception() (in module traceback), 1276
PRINT_EXPR (opcode), 1345
print_form() (in module cgi), 918
print_help() (argparse.ArgumentParser method), 490
PRINT_ITEM (opcode), 1345
PRINT_ITEM_TO (opcode), 1345
print_last() (in module traceback), 1276
PRINT_NEWLINE (opcode), 1346
PRINT_NEWLINE_TO (opcode), 1346
print_stack() (in module traceback), 1276
print_stats() (profile.Profile method), 1228
print_stats() (pstats.Stats method), 1230
print_tb() (in module traceback), 1276
print_usage() (argparse.ArgumentParser method), 490
print_usage() (optparse.OptionParser method), 512
print_version() (optparse.OptionParser method), 501
printable (in module string), 76
printdir() (zipfile.ZipFile method), 373
printf-style formatting, 46
PriorityQueue (class in Queue), 204
prmonth() (calendar.TextCalendar method), 171
prmonth() (in module calendar), 173
process
  group, 416, 417
  id, 417
  id of parent, 417
  killing, 438
  signalling, 438
Process (class in multiprocessing), 644
process() (logging.LoggerAdapter method), 531
process_message() (smtpd.SMTPServer method), 977
process_request() (SocketServer.BaseServer method), 992
process_tokens() (in module tabnanny), 1337
ProcessError, 666
processes, light-weight, 637
processfile() (in module gensuitemodule), 1420
processfile_fromresource() (in module gensuitemodule), 1420
ProcessingInstruction() (in module xml.etree.ElementTree), 866
processingInstruction() (xml.sax.handler.ContentHandler method), 894
ProcessingInstructionHandler() (xml.parsers.expat.xmlparser method), 904
processor time, 457
processor() (in module platform), 577
product() (in module itertools), 274
Profile (class in hotshot), 1233
Profile (class in profile), 1227
profile (module), 1227
profile function, 628, 1252, 1257
profiler, 1252, 1257
profiling, deterministic, 1224
Progressbar (class in ttk), 1094
ProgressBar() (in module EasyDialogs), 1408
prompt (cmd.Cmd attribute), 1069
prompt_user_passwd() (urllib.FancyURLopener method), 936
prompts, interpreter, 1256
propagate (logging.Logger attribute), 522
property (built-in class), 16
property list, 407
property_declaration_handler (in module xml.sax.handler), 891
property_dom_node (in module xml.sax.handler), 892
property_lexical_handler (in module xml.sax.handler), 891
property_xml_string (in module xml.sax.handler), 892
prot_c() (ftplib.FTP_TLS method), 960
prot_p() (ftplib.FTP_TLS method), 960
proto (socket.socket attribute), 723
protocol
  CGI, 914
  context management, 62
  FTP, 936, 956
  HTTP, 914, 936, 949, 996
  IMAP4, 962
  IMAP4_SSL, 962
  IMAP4_stream, 962
  iterator, 37
  NNTP, 968
  POP3, 960
  SMTP, 972
  Telnet, 978
protocol (ssl.SSLContext attribute), 743
PROTOCOL_SSLv2 (in module ssl), 733
PROTOCOL_SSLv23 (in module ssl), 733
PROTOCOL_SSLv3 (in module ssl), 733
PROTOCOL_TLS (in module ssl), 733
PROTOCOL_TLSv1 (in module ssl), 734

PROTOCOL_TLSv1_1 (in module ssl), 734
PROTOCOL_TLSv1_2 (in module ssl), 734
protocol_version (BaseHTTPServer.BaseHTTPRequestHandler attribute), 998
PROTOCOL_VERSION (imaplib.IMAP4 attribute), 968
ProtocolError (class in xmlrpclib), 1020
proxy() (in module weakref), 207
proxyauth() (imaplib.IMAP4 method), 966
ProxyBasicAuthHandler (class in urllib2), 940
ProxyDigestAuthHandler (class in urllib2), 941
ProxyHandler (class in urllib2), 940
ProxyType (in module weakref), 209
ProxyTypes (in module weakref), 209
prstr() (in module fm), 1438
pryear() (calendar.TextCalendar method), 172
ps1 (in module sys), 1256
ps2 (in module sys), 1256
pstats (module), 1228
pthreads, 637
ptime (in module cd), 1430
pty
    module, 424
pty (module), 1390
pu() (in module turtle), 1120
publicId (xml.dom.DocumentType attribute), 877
PullDOM (class in xml.dom.pulldom), 888
punctuation (in module string), 76
PureProxy (class in smtpd), 978
purge() (in module re), 97
Purpose.CLIENT_AUTH (in module ssl), 736
Purpose.SERVER_AUTH (in module ssl), 736
push() (asynchat.async_chat method), 761
push() (asynchat.fifo method), 762
push() (code.InteractiveConsole method), 1296
push() (multifile.MultiFile method), 835
push_alignment() (formatter.formatter method), 1360
push_font() (formatter.formatter method), 1360
push_margin() (formatter.formatter method), 1360
push_source() (shlex.shlex method), 1071
push_style() (formatter.formatter method), 1361
push_token() (shlex.shlex method), 1070
push_with_producer() (asynchat.async_chat method), 761
pushbutton() (msilib.Dialog method), 1370
put() (multiprocessing.multiprocessing.queues.SimpleQueue method), 649
put() (multiprocessing.Queue method), 648
put() (Queue.Queue method), 205
put_nowait() (multiprocessing.Queue method), 648
put_nowait() (Queue.Queue method), 205
putch() (in module msvcrt), 1372
putenv() (in module os), 418
putheader() (httplib.HTTPConnection method), 954
putp() (in module curses), 558
putrequest() (httplib.HTTPConnection method), 953
putsequences() (mhlib.Folder method), 827
putwch() (in module msvcrt), 1372
putwin() (curses.window method), 565
pwd
    module, 292
pwd (module), 1384
pwd() (ftplib.FTP method), 959
pwlcurve() (in module gl), 1440
py3kwarning (in module sys), 1256
py_compile (module), 1339
PY_COMPILED (in module imp), 1306
PY_FROZEN (in module imp), 1307
py_object (class in ctypes), 617
PY_SOURCE (in module imp), 1306
py_suffix_importer() (in module imputil), 1310
pyclbr (module), 1337
PyCompileError, 1339
PyDLL (class in ctypes), 606
pydoc (module), 1151
pyexpat
    module, 900
PYFUNCTYPE() (in module ctypes), 609
PyOpenGL, 1441
Python 3000, 1458
Python Editor, 1140
Python Enhancement Proposals
    PEP 205, 209
    PEP 227, 1280
    PEP 238, 1280
    PEP 249, 343
    PEP 255, 1280
    PEP 273, 1313
    PEP 282, 534
    PEP 302, 1313
    PEP 305, 387
    PEP 3105, 1280
    PEP 3112, 1280
    PEP 328, 1280
    PEP 338, 1320
    PEP 343, 1271, 1280
    PEP 366, 1320
    PEP 453, 1244
    PEP 477, 1244
Python Enhancement Proposals
    PEP 1, 1458
    PEP 236, 7
    PEP 237, 48
    PEP 238, 1454
    PEP 249, 342
    PEP 278, 1459

PEP 282, 313
PEP 292, 84
PEP 302, 25, 309, 1255, 1308, 1315, 1316, 1318, 1319, 1453, 1456
PEP 307, 318
PEP 3101, 76, 77
PEP 3116, 1459
PEP 3119, 189, 1271
PEP 3141, 223, 1271
PEP 324, 701
PEP 333, 922–926, 929
PEP 343, 1452
PEP 370, 1292
PEP 378, 80
PEP 476, 730
PEP 493, 730
PEP 8, 626, 794
Python Imaging Library, 1442
python_branch() (in module platform), 577
python_build() (in module platform), 577
python_compiler() (in module platform), 577
PYTHON_DOM, 873
python_implementation() (in module platform), 577
python_revision() (in module platform), 577
python_version() (in module platform), 578
python_version_tuple() (in module platform), 578
PYTHONDOCS, 1152
PYTHONDONTWRITEBYTECODE, 1248
PYTHONHASHSEED, 263
Pythonic, 1458
PYTHONNOUSERSITE, 1291
PYTHONPATH, 919, 1255
PYTHONSTARTUP, 698, 700, 1146, 1292
PYTHONUSERBASE, 1291
PYTHONY2K, 456, 457
PyZipFile (class in zipfile), 371

Q

qdevice() (in module fl), 1434
QDPoint (class in aetypes), 1423
QDRectangle (class in aetypes), 1423
qenter() (in module fl), 1434
qiflush() (in module curses), 559
QName (class in xml.etree.ElementTree), 870
qread() (in module fl), 1434
qreset() (in module fl), 1434
qsize() (multiprocessing.Queue method), 648
qsize() (Queue.Queue method), 205
qtest() (in module fl), 1434
quantize() (decimal.Context method), 251
quantize() (decimal.Decimal method), 243
QueryInfoKey() (in module _winreg), 1375
queryparams() (in module al), 1427
QueryReflectionKey() (in module _winreg), 1377
QueryValue() (in module _winreg), 1375
QueryValueEx() (in module _winreg), 1376
Queue (class in multiprocessing), 648
Queue (class in Queue), 204
Queue (module), 204
queue (sched.scheduler attribute), 203
Queue() (multiprocessing.managers.SyncManager method), 659
quick_ratio() (difflib.SequenceMatcher method), 116
quit (built-in variable), 29
quit() (ftplib.FTP method), 960
quit() (nntplib.NNTP method), 972
quit() (poplib.POP3 method), 962
quit() (smtplib.SMTP method), 976
quopri (module), 844
quote() (in module email.utils), 785
quote() (in module pipes), 1393
quote() (in module rfc822), 837
quote() (in module urllib), 933
QUOTE_ALL (in module csv), 390
QUOTE_MINIMAL (in module csv), 390
QUOTE_NONE (in module csv), 390
QUOTE_NONNUMERIC (in module csv), 390
quote_plus() (in module urllib), 933
quoteattr() (in module xml.sax.saxutils), 895
quotechar (csv.Dialect attribute), 391
quoted-printable
    encoding, 844
quotes (shlex.shlex attribute), 1071
quoting (csv.Dialect attribute), 391

R

r_eval() (rexec.RExec method), 1300
r_exec() (rexec.RExec method), 1301
r_execfile() (rexec.RExec method), 1301
r_import() (rexec.RExec method), 1301
R_OK (in module os), 426
r_open() (rexec.RExec method), 1301
r_reload() (rexec.RExec method), 1301
r_unload() (rexec.RExec method), 1301
radians() (in module math), 229
radians() (in module turtle), 1120
RadioButtonGroup (class in msilib), 1369
radiogroup() (msilib.Dialog method), 1370
radix() (decimal.Context method), 251
radix() (decimal.Decimal method), 244
RADIXCHAR (in module locale), 1061
raise
    statement, 67
raise (2to3 fixer), 1206
RAISE_VARARGS (opcode), 1349
RAND_add() (in module ssl), 730
RAND_egd() (in module ssl), 730
RAND_status() (in module ssl), 730

randint() (in module random), 263
random (module), 262
random() (in module random), 264
randrange() (in module random), 263
Range (class in aetypes), 1424
range() (built-in function), 18
ratecv() (in module audioop), 1031
ratio() (difflib.SequenceMatcher method), 116
Rational (class in numbers), 223
raw (io.BufferedIOBase attribute), 450
raw() (in module curses), 559
raw_decode() (json.JSONDecoder method), 802
raw_input
    built-in function, 1258
raw_input (2to3 fixer), 1206
raw_input() (built-in function), 18
raw_input() (code.InteractiveConsole method), 1297
RawArray() (in module multiprocessing.sharedctypes), 655
RawConfigParser (class in ConfigParser), 396
RawDescriptionHelpFormatter (class in argparse), 467
RawIOBase (class in io), 449
RawPen (class in turtle), 1136
RawTextHelpFormatter (class in argparse), 467
RawTurtle (class in turtle), 1136
RawValue() (in module multiprocessing.sharedctypes), 655
RBRACE (in module token), 1333
re
    module, 48, 75, 308
re (module), 88
re (re.MatchObject attribute), 100
read() (array.array method), 197
read() (bz2.BZ2File method), 369
read() (chunk.Chunk method), 1041
read() (codecs.StreamReader method), 132
read() (ConfigParser.RawConfigParser method), 398
read() (file method), 58
read() (httplib.HTTPResponse method), 954
read() (imaplib.IMAP4 method), 966
read() (in module imgfile), 1441
read() (in module os), 424
read() (io.BufferedIOBase method), 450
read() (io.BufferedReader method), 452
read() (io.RawIOBase method), 449
read() (io.TextIOBase method), 453
read() (mimetypes.MimeTypes method), 831
read() (mmap.mmap method), 695
read() (multifile.MultiFile method), 834
read() (ossaudiodev.oss_audio_device method), 1045
read() (robotparser.RobotFileParser method), 402
read() (zipfile.ZipFile method), 373
read1() (io.BufferedIOBase method), 450
read1() (io.BufferedReader method), 452
read1() (io.BytesIO method), 451
read_all() (telnetlib.Telnet method), 979
read_byte() (mmap.mmap method), 695
read_eager() (telnetlib.Telnet method), 979
read_history_file() (in module readline), 696
read_init_file() (in module readline), 696
read_lazy() (telnetlib.Telnet method), 979
read_mime_types() (in module mimetypes), 830
read_sb_data() (telnetlib.Telnet method), 979
read_some() (telnetlib.Telnet method), 979
read_token() (shlex.shlex method), 1070
read_until() (telnetlib.Telnet method), 979
read_very_eager() (telnetlib.Telnet method), 979
read_very_lazy() (telnetlib.Telnet method), 979
read_windows_registry() (mimetypes.MimeTypes method), 832
READABLE (in module Tkinter), 1086
readable() (asyncore.dispatcher method), 758
readable() (io.IOBase method), 448
readall() (io.RawIOBase method), 449
reader() (in module csv), 387
ReadError, 378
readfp() (ConfigParser.RawConfigParser method), 398
readfp() (mimetypes.MimeTypes method), 831
readframes() (aifc.aifc method), 1034
readframes() (sunau.AU_read method), 1037
readframes() (wave.Wave_read method), 1039
readinto() (io.BufferedIOBase method), 450
readinto() (io.RawIOBase method), 449
readline (module), 695
readline() (bz2.BZ2File method), 369
readline() (codecs.StreamReader method), 132
readline() (file method), 59
readline() (imaplib.IMAP4 method), 966
readline() (io.IOBase method), 448
readline() (io.TextIOBase method), 453
readline() (mmap.mmap method), 695
readline() (multifile.MultiFile method), 834
readlines() (bz2.BZ2File method), 369
readlines() (codecs.StreamReader method), 132
readlines() (file method), 59
readlines() (io.IOBase method), 448
readlines() (multifile.MultiFile method), 834
readlink() (in module os), 430
readmodule() (in module pyclbr), 1337
readmodule_ex() (in module pyclbr), 1338
readonly (memoryview attribute), 62
readPlist() (in module plistlib), 407
readPlistFromResource() (in module plistlib), 407
readPlistFromString() (in module plistlib), 407
readscaled() (in module imgfile), 1441

READY (in module cd), 1430
ready() (multiprocessing.pool.AsyncResult method), 665
Real (class in numbers), 223
real (numbers.Complex attribute), 223
Real Media File Format, 1040
real_quick_ratio() (difflib.SequenceMatcher method), 116
realpath() (in module os.path), 293
reason (exceptions.UnicodeError attribute), 71
reason (httplib.HTTPResponse attribute), 954
reason (ssl.SSLError attribute), 727
reason (urllib2.HTTPError attribute), 939
reason (urllib2.URLError attribute), 939
reattach() (ttk.Treeview method), 1100
reccontrols() (ossaudiodev.oss_mixer_device method), 1047
recent() (imaplib.IMAP4 method), 966
rect() (in module cmath), 231
rectangle() (in module curses.textpad), 571
recv() (asyncore.dispatcher method), 758
recv() (Connection method), 650
recv() (socket.socket method), 721
recv_bytes() (Connection method), 651
recv_bytes_into() (Connection method), 651
recv_into() (socket.socket method), 721
recvfrom() (socket.socket method), 721
recvfrom_into() (socket.socket method), 721
redirect_request() (urllib2.HTTPRedirectHandler method), 945
redisplay() (in module readline), 696
redraw_form() (fl.form method), 1435
redrawln() (curses.window method), 565
redrawwin() (curses.window method), 565
reduce (2to3 fixer), 1206
reduce() (built-in function), 18
reduce() (in module functools), 280
ref (class in weakref), 207
reference count, 1458
ReferenceError, 69, 209
ReferenceType (in module weakref), 209
refilemessages() (mhlib.Folder method), 827
refresh() (curses.window method), 565
REG_BINARY (in module _winreg), 1379
REG_DWORD (in module _winreg), 1379
REG_DWORD_BIG_ENDIAN (in module _winreg), 1379
REG_DWORD_LITTLE_ENDIAN (in module _winreg), 1379
REG_EXPAND_SZ (in module _winreg), 1379
REG_FULL_RESOURCE_DESCRIPTOR (in module _winreg), 1379
REG_LINK (in module _winreg), 1379
REG_MULTI_SZ (in module _winreg), 1379
REG_NONE (in module _winreg), 1379
REG_RESOURCE_LIST (in module _winreg), 1379
REG_RESOURCE_REQUIREMENTS_LIST (in module _winreg), 1379
REG_SZ (in module _winreg), 1379
RegexObject (class in re), 97
register() (abc.ABCMeta method), 1272
register() (in module atexit), 1274
register() (in module codecs), 125
register() (in module webbrowser), 912
register() (multiprocessing.managers.BaseManager method), 658
register() (select.epoll method), 623
register() (select.poll method), 623
register_adapter() (in module sqlite3), 344
register_archive_format() (in module shutil), 313
register_converter() (in module sqlite3), 344
register_dialect() (in module csv), 388
register_error() (in module codecs), 126
register_function() (SimpleXMLRPCServer.CGIXMLRPCRequestHa method), 1026
register_function() (SimpleXMLRPCServer.SimpleXMLRPCServer method), 1024
register_instance() (SimpleXMLRPCServer.CGIXMLRPCRequestHa method), 1026
register_instance() (SimpleXMLRPCServer.SimpleXMLRPCServer method), 1024
register_introspection_functions() (SimpleXMLRPCServer.CGIXMLRPCRequestHandler method), 1027
register_introspection_functions() (SimpleXMLRPCServer.SimpleXMLRPCServer method), 1024
register_multicall_functions() (SimpleXMLRPCServer.CGIXMLRPCRequestHandler method), 1027
register_multicall_functions() (SimpleXMLRPCServer.SimpleXMLRPCServer method), 1024
register_namespace() (in module xml.etree.ElementTree), 866
register_optionflag() (in module doctest), 1161
register_shape() (in module turtle), 1134
registerDOMImplementation() (in module xml.dom), 873
registerResult() (in module unittest), 1202
relative URL, 984
release() (in module platform), 578
release() (logging.Handler method), 525
release() (multiprocessing.Lock method), 653
release() (multiprocessing.RLock method), 653
release() (thread.lock method), 638

release() (threading.Condition method), 633
release() (threading.Lock method), 631
release() (threading.RLock method), 632
release() (threading.Semaphore method), 634
release_lock() (in module imp), 1306
reload
    built-in function, 1255, 1306, 1308
reload() (built-in function), 19
relpath() (in module os.path), 293
remainder() (decimal.Context method), 251
remainder_near() (decimal.Context method), 251
remainder_near() (decimal.Decimal method), 244
remove() (array.array method), 197
remove() (collections.deque method), 178
remove() (frozenset method), 52
remove() (in module os), 430
remove() (list method), 48
remove() (mailbox.Mailbox method), 807
remove() (mailbox.MH method), 812
remove() (xml.etree.ElementTree.Element method), 869
remove_flag() (mailbox.MaildirMessage method), 816
remove_flag() (mailbox.mboxMessage method), 817
remove_flag() (mailbox.MMDFMessage method), 821
remove_folder() (mailbox.Maildir method), 810
remove_folder() (mailbox.MH method), 812
remove_history_item() (in module readline), 697
remove_label() (mailbox.BabylMessage method), 820
remove_option() (ConfigParser.RawConfigParser method), 399
remove_option() (optparse.OptionParser method), 510
remove_pyc() (msilib.Directory method), 1369
remove_section() (ConfigParser.RawConfigParser method), 399
remove_sequence() (mailbox.MHMessage method), 819
removeAttribute() (xml.dom.Element method), 879
removeAttributeNode() (xml.dom.Element method), 879
removeAttributeNS() (xml.dom.Element method), 879
removeChild() (xml.dom.Node method), 876
removedirs() (in module os), 430
removeFilter() (logging.Handler method), 526
removeFilter() (logging.Logger method), 524
removeHandler() (in module unittest), 1202
removeHandler() (logging.Logger method), 525
removemessages() (mhlib.Folder method), 827
removeResult() (in module unittest), 1202
rename() (ftplib.FTP method), 959
rename() (imaplib.IMAP4 method), 966
rename() (in module os), 430
renames (2to3 fixer), 1206
renames() (in module os), 430
reorganize() (in module gdbm), 337
repeat() (in module itertools), 275
repeat() (in module operator), 285
repeat() (in module timeit), 1235
repeat() (timeit.Timer method), 1236
repetition
    operation, 38
replace() (curses.panel.Panel method), 576
replace() (datetime.date method), 150
replace() (datetime.datetime method), 155
replace() (datetime.time method), 160
replace() (in module string), 88
replace() (str method), 42
replace_errors() (in module codecs), 127
replace_header() (email.message.Message method), 769
replace_history_item() (in module readline), 697
replace_whitespace (textwrap.TextWrapper attribute), 123
replaceChild() (xml.dom.Node method), 876
ReplacePackage() (in module modulefinder), 1317
report() (filecmp.dircmp method), 303
report() (modulefinder.ModuleFinder method), 1317
REPORT_CDIFF (in module doctest), 1161
report_failure() (doctest.DocTestRunner method), 1172
report_full_closure() (filecmp.dircmp method), 303
REPORT_NDIFF (in module doctest), 1161
REPORT_ONLY_FIRST_FAILURE (in module doctest), 1161
report_partial_closure() (filecmp.dircmp method), 303
report_start() (doctest.DocTestRunner method), 1171
report_success() (doctest.DocTestRunner method), 1172
REPORT_UDIFF (in module doctest), 1161
report_unbalanced() (sgmllib.SGMLParser method), 854
report_unexpected_exception() (doctest.DocTestRunner method), 1172
REPORTING_FLAGS (in module doctest), 1161
repr (2to3 fixer), 1207
Repr (class in repr), 221
repr() (built-in function), 19
repr() (in module repr), 221
repr() (repr.Repr method), 221
repr1() (repr.Repr method), 221
Request (class in urllib2), 939
request() (httplib.HTTPConnection method), 953

request_queue_size (SocketServer.BaseServer attribute), 991
request_uri() (in module wsgiref.util), 922
request_version (BaseHTTPServer.BaseHTTPRequestHandler attribute), 997
RequestHandlerClass (SocketServer.BaseServer attribute), 991
requires() (in module test.support), 1211
reserved (zipfile.ZipInfo attribute), 375
RESERVED_FUTURE (in module uuid), 983
RESERVED_MICROSOFT (in module uuid), 983
RESERVED_NCS (in module uuid), 983
reset() (bdb.Bdb method), 1216
reset() (codecs.IncrementalDecoder method), 130
reset() (codecs.IncrementalEncoder method), 130
reset() (codecs.StreamReader method), 132
reset() (codecs.StreamWriter method), 131
reset() (HTMLParser.HTMLParser method), 848
reset() (in module dircache), 314
reset() (in module turtle), 1124, 1130
reset() (ossaudiodev.oss_audio_device method), 1046
reset() (pipes.Template method), 1393
reset() (sgmllib.SGMLParser method), 852
reset() (xdrlib.Packer method), 404
reset() (xdrlib.Unpacker method), 405
reset() (xml.dom.pulldom.DOMEventStream method), 888
reset() (xml.sax.xmlreader.IncrementalParser method), 898
reset_prog_mode() (in module curses), 559
reset_shell_mode() (in module curses), 559
resetbuffer() (code.InteractiveConsole method), 1297
resetlocale() (in module locale), 1063
resetscreen() (in module turtle), 1130
resetty() (in module curses), 559
resetwarnings() (in module warnings), 1269
resize() (curses.window method), 566
resize() (in module ctypes), 613
resize() (mmap.mmap method), 695
resize_term() (in module curses), 559
resizemode() (in module turtle), 1125
resizeterm() (in module curses), 559
resolution (datetime.date attribute), 150
resolution (datetime.datetime attribute), 154
resolution (datetime.time attribute), 159
resolution (datetime.timedelta attribute), 147
resolveEntity() (xml.sax.handler.EntityResolver method), 894
resource (module), 1396
ResourceDenied, 1210
response() (imaplib.IMAP4 method), 966
ResponseNotReady, 951
responses (BaseHTTPServer.BaseHTTPRequestHandler attribute), 998
responses (in module httplib), 952
Result() (in module findertools), 1407
restore() (in module difflib), 113
restype (ctypes._FuncPtr attribute), 608
results() (trace.Trace method), 1241
retr() (poplib.POP3 method), 961
retrbinary() (ftplib.FTP method), 958
retrieve() (urllib.URLopener method), 935
retrlines() (ftplib.FTP method), 958
return_ok() (cookielib.CookiePolicy method), 1006
RETURN_VALUE (opcode), 1346
returncode (subprocess.CalledProcessError attribute), 703
returncode (subprocess.Popen attribute), 708
returns_unicode (xml.parsers.expat.xmlparser attribute), 903
reverse() (array.array method), 197
reverse() (collections.deque method), 178
reverse() (in module audioop), 1031
reverse() (list method), 48
reverse_order() (pstats.Stats method), 1230
reversed() (built-in function), 20
revert() (cookielib.FileCookieJar method), 1005
rewind() (aifc.aifc method), 1034
rewind() (sunau.AU_read method), 1037
rewind() (wave.Wave_read method), 1039
rewindbody() (rfc822.Message method), 838
RExec (class in rexec), 1300
rexec (module), 1299
RFC
    RFC 1014, 404
    RFC 1521, 842
    RFC 1738, 987
    RFC 1808, 987
    RFC 1832, 404
    RFC 1869, 974
    RFC 2109, 1004, 1012
    RFC 2368, 987
    RFC 2396, 987
    RFC 2732, 987
    RFC 2965, 1004
    RFC 3490, 138
    RFC 3492, 138
    RFC 3986, 987
    RFC 4122, 983
    RFC 821, 974
    RFC 854, 978
RFC
    RFC 1014, 404
    RFC 1123, 459
    RFC 1321, 409, 413
    RFC 1422, 744

RFC 1521, 844, 845
RFC 1522, 845
RFC 1524, 805
RFC 1725, 960
RFC 1730, 963
RFC 1750, 730
RFC 1766, 1062
RFC 1808, 984
RFC 1832, 404
RFC 1866, 855
RFC 1869, 972
RFC 1894, 796
RFC 2045, 765, 769, 770, 779, 835
RFC 2046, 765, 779
RFC 2047, 765, 779, 780
RFC 2060, 963, 967
RFC 2068, 1011
RFC 2104, 412
RFC 2109, 1002, 1011, 1013
RFC 2231, 765, 769, 770, 779, 786, 787, 794
RFC 2396, 986
RFC 2616, 923, 935, 945
RFC 2774, 952
RFC 2817, 952
RFC 2818, 730
RFC 2821, 765
RFC 2822, 459, 765–767, 774–776, 779, 780, 784–786, 814, 836–838, 977
RFC 2964, 1004
RFC 2965, 939, 942, 1002
RFC 3229, 952
RFC 3280, 737
RFC 3454, 141
RFC 3490, 138, 139
RFC 3492, 138
RFC 3493, 713
RFC 3548, 840, 841, 843
RFC 4122, 981, 983
RFC 4217, 956
RFC 4366, 735
RFC 4627, 797, 805
RFC 5246, 736
RFC 5280, 731
RFC 5929, 738
RFC 6066, 741
RFC 6125, 730
RFC 7159, 797, 803, 805
RFC 7301, 735, 741
RFC 821, 972
RFC 822, 395, 459, 779, 836, 954, 975, 976, 1053
RFC 854, 978
RFC 959, 956
RFC 977, 968
rfc2109 (cookielib.Cookie attribute), 1010
rfc2109_as_netscape (cookielib.DefaultCookiePolicy attribute), 1008
rfc2965 (cookielib.CookiePolicy attribute), 1007
rfc822
    module, 828
rfc822 (module), 836
RFC_4122 (in module uuid), 983
rfile (BaseHTTPServer.BaseHTTPRequestHandler attribute), 997
rfind() (in module string), 86
rfind() (mmap.mmap method), 695
rfind() (str method), 42
rgb_to_hls() (in module colorsys), 1042
rgb_to_hsv() (in module colorsys), 1042
rgb_to_yiq() (in module colorsys), 1042
RGBColor (class in aetypes), 1423
right (filecmp.dircmp attribute), 303
right() (in module turtle), 1114
right_list (filecmp.dircmp attribute), 303
right_only (filecmp.dircmp attribute), 303
RIGHTSHIFT (in module token), 1333
RIGHTSHIFTEQUAL (in module token), 1333
rindex() (in module string), 86
rindex() (str method), 42
rjust() (in module string), 88
rjust() (str method), 42
rlcompleter (module), 699
rlecode_hqx() (in module binascii), 843
rledecode_hqx() (in module binascii), 843
RLIM_INFINITY (in module resource), 1396
RLIMIT_AS (in module resource), 1397
RLIMIT_CORE (in module resource), 1397
RLIMIT_CPU (in module resource), 1397
RLIMIT_DATA (in module resource), 1397
RLIMIT_FSIZE (in module resource), 1397
RLIMIT_MEMLOCK (in module resource), 1397
RLIMIT_NOFILE (in module resource), 1397
RLIMIT_NPROC (in module resource), 1397
RLIMIT_OFILE (in module resource), 1397
RLIMIT_RSS (in module resource), 1397
RLIMIT_STACK (in module resource), 1397
RLIMIT_VMEM (in module resource), 1397
RLock (class in multiprocessing), 653
RLock() (in module threading), 628
RLock() (multiprocessing.managers.SyncManager method), 659
rmd() (ftplib.FTP method), 959
rmdir() (in module os), 431
RMFF, 1040
rms() (in module audioop), 1031
rmtree() (in module shutil), 311
rnopen() (in module bsddb), 339
RobotFileParser (class in robotparser), 402
robotparser (module), 402

robots.txt, 402
rollback() (sqlite3.Connection method), 346
ROT_FOUR (opcode), 1343
ROT_THREE (opcode), 1343
ROT_TWO (opcode), 1343
rotate() (collections.deque method), 178
rotate() (decimal.Context method), 251
rotate() (decimal.Decimal method), 244
RotatingFileHandler (class in logging.handlers), 546
round() (built-in function), 20
Rounded (class in decimal), 252
Row (class in sqlite3), 353
row_factory (sqlite3.Connection attribute), 349
rowcount (sqlite3.Cursor attribute), 353
RPAR (in module token), 1333
rpartition() (str method), 43
rpc_paths (SimpleXMLRPCServer.SimpleXMLRPCRequestHandler attribute), 1025
rpop() (poplib.POP3 method), 961
rset() (poplib.POP3 method), 962
rshift() (in module operator), 284
rsplit() (in module string), 87
rsplit() (str method), 43
RSQB (in module token), 1333
rstrip() (in module string), 87
rstrip() (str method), 43
rt() (in module turtle), 1114
RTLD_LAZY (in module dl), 1387
RTLD_NOW (in module dl), 1387
ruler (cmd.Cmd attribute), 1069
Run script, 1142
run() (bdb.Bdb method), 1219
run() (doctest.DocTestRunner method), 1172
run() (hotshot.Profile method), 1233
run() (in module pdb), 1220
run() (in module profile), 1227
run() (multiprocessing.Process method), 645
run() (pdb.Pdb method), 1221
run() (profile.Profile method), 1228
run() (sched.scheduler method), 203
run() (threading.Thread method), 630
run() (trace.Trace method), 1240
run() (unittest.TestCase method), 1187
run() (unittest.TestSuite method), 1194
run() (wsgiref.handlers.BaseHandler method), 928
run_docstring_examples() (in module doctest), 1165
run_module() (in module runpy), 1318
run_path() (in module runpy), 1319
run_script() (modulefinder.ModuleFinder method), 1317
run_unittest() (in module test.support), 1211
runcall() (bdb.Bdb method), 1219
runcall() (hotshot.Profile method), 1233
runcall() (in module pdb), 1221
runcall() (pdb.Pdb method), 1221
runcall() (profile.Profile method), 1228
runcode() (code.InteractiveInterpreter method), 1296
runctx() (bdb.Bdb method), 1219
runctx() (hotshot.Profile method), 1233
runctx() (in module profile), 1227
runctx() (profile.Profile method), 1228
runctx() (trace.Trace method), 1240
runeval() (bdb.Bdb method), 1219
runeval() (in module pdb), 1220
runeval() (pdb.Pdb method), 1221
runfunc() (trace.Trace method), 1240
runpy (module), 1318
runsource() (code.InteractiveInterpreter method), 1296
RuntimeError, 70
RuntimeHandler (in module MacOS), 1405
RuntimeWarning, 72
RUSAGE_BOTH (in module resource), 1398
RUSAGE_CHILDREN (in module resource), 1398
RUSAGE_SELF (in module resource), 1398

S

S (in module re), 94
S_ENFMT (in module stat), 300
s_eval() (rexec.RExec method), 1301
s_exec() (rexec.RExec method), 1301
s_execfile() (rexec.RExec method), 1301
S_IEXEC (in module stat), 300
S_IFBLK (in module stat), 299
S_IFCHR (in module stat), 299
S_IFDIR (in module stat), 299
S_IFIFO (in module stat), 299
S_IFLNK (in module stat), 299
S_IFMT() (in module stat), 298
S_IFREG (in module stat), 299
S_IFSOCK (in module stat), 299
S_IMODE() (in module stat), 298
s_import() (rexec.RExec method), 1301
S_IREAD (in module stat), 300
S_IRGRP (in module stat), 300
S_IROTH (in module stat), 300
S_IRUSR (in module stat), 300
S_IRWXG (in module stat), 300
S_IRWXO (in module stat), 300
S_IRWXU (in module stat), 300
S_ISBLK() (in module stat), 297
S_ISCHR() (in module stat), 297
S_ISDIR() (in module stat), 297
S_ISFIFO() (in module stat), 297
S_ISGID (in module stat), 299
S_ISLNK() (in module stat), 297
S_ISREG() (in module stat), 297
S_ISSOCK() (in module stat), 298

S_ISUID (in module stat), 299
S_ISVTX (in module stat), 300
S_IWGRP (in module stat), 300
S_IWOTH (in module stat), 300
S_IWRITE (in module stat), 300
S_IWUSR (in module stat), 300
S_IXGRP (in module stat), 300
S_IXOTH (in module stat), 300
S_IXUSR (in module stat), 300
s_reload() (rexec.RExec method), 1301
s_unload() (rexec.RExec method), 1301
safe_substitute() (string.Template method), 84
SafeConfigParser (class in ConfigParser), 396
saferepr() (in module pprint), 219
same_files (filecmp.dircmp attribute), 303
same_quantum() (decimal.Context method), 251
same_quantum() (decimal.Decimal method), 244
samefile() (in module os.path), 294
sameopenfile() (in module os.path), 294
samestat() (in module os.path), 294
sample() (in module random), 264
save() (cookielib.FileCookieJar method), 1005
save_bgn() (htmllib.HTMLParser method), 856
save_end() (htmllib.HTMLParser method), 856
SaveKey() (in module _winreg), 1376
savetty() (in module curses), 559
SAX2DOM (class in xml.dom.pulldom), 888
SAXException, 889
SAXNotRecognizedException, 889
SAXNotSupportedException, 889
SAXParseException, 889
scale() (in module imageop), 1032
scaleb() (decimal.Context method), 251
scaleb() (decimal.Decimal method), 244
scalebarvalues() (FrameWork.ScrolledWindow method), 1413
scanf(), 102
sched (module), 201
scheduler (class in sched), 201
schema (in module msilib), 1370
sci() (in module fpformat), 143
Scrap Manager, 1416
Screen (class in turtle), 1136
screensize() (in module turtle), 1131
script_from_examples() (in module doctest), 1174
scroll() (curses.window method), 566
scrollbar_callback() (FrameWork.ScrolledWindow method), 1413
scrollbars() (FrameWork.ScrolledWindow method), 1413
ScrolledCanvas (class in turtle), 1136
ScrolledText (module), 1109
scrollok() (curses.window method), 566
search
    path, module, 309, 1255, 1290
search() (imaplib.IMAP4 method), 966
search() (in module re), 94
search() (re.RegexObject method), 97
SEARCH_ERROR (in module imp), 1307
second (datetime.datetime attribute), 154
second (datetime.time attribute), 160
section_divider() (multifile.MultiFile method), 835
sections() (ConfigParser.RawConfigParser method), 397
secure (cookielib.Cookie attribute), 1009
Secure Hash Algorithm, 414
secure hash algorithm, SHA1, SHA224, SHA256, SHA384, SHA512, 409
Secure Sockets Layer, 726
security
    CGI, 918
see() (ttk.Treeview method), 1100
seed() (in module random), 263
seek() (bz2.BZ2File method), 369
seek() (chunk.Chunk method), 1041
seek() (file method), 59
seek() (io.IOBase method), 448
seek() (io.TextIOBase method), 453
seek() (mmap.mmap method), 695
seek() (multifile.MultiFile method), 834
SEEK_CUR (in module os), 423
SEEK_CUR (in module posixfile), 1394
SEEK_END (in module os), 423
SEEK_END (in module posixfile), 1394
SEEK_SET (in module os), 423
SEEK_SET (in module posixfile), 1394
seekable() (io.IOBase method), 449
Select (class in Tix), 1106
select (module), 621
select() (imaplib.IMAP4 method), 966
select() (in module gl), 1440
select() (in module select), 621
select() (ttk.Notebook method), 1093
selected_alpn_protocol() (ssl.SSLSocket method), 739
selected_npn_protocol() (ssl.SSLSocket method), 739
selection() (ttk.Treeview method), 1100
selection_add() (ttk.Treeview method), 1100
selection_remove() (ttk.Treeview method), 1100
selection_set() (ttk.Treeview method), 1100
selection_toggle() (ttk.Treeview method), 1100
Semaphore (class in multiprocessing), 653
Semaphore (class in threading), 634
Semaphore() (multiprocessing.managers.SyncManager method), 659
semaphores, binary, 637
SEMI (in module token), 1333

send() (aetools.TalkTo method), 1421
send() (asyncore.dispatcher method), 758
send() (Connection method), 650
send() (httplib.HTTPConnection method), 954
send() (imaplib.IMAP4 method), 966
send() (logging.handlers.DatagramHandler method), 549
send() (logging.handlers.SocketHandler method), 548
send() (socket.socket method), 721
send_bytes() (Connection method), 651
send_error() (BaseHTTPServer.BaseHTTPRequestHandler method), 998
send_flowing_data() (formatter.writer method), 1362
send_header() (BaseHTTPServer.BaseHTTPRequestHandler method), 998
send_hor_rule() (formatter.writer method), 1362
send_label_data() (formatter.writer method), 1362
send_line_break() (formatter.writer method), 1362
send_literal_data() (formatter.writer method), 1362
send_paragraph() (formatter.writer method), 1362
send_response() (BaseHTTPServer.BaseHTTPRequestHandler method), 998
send_signal() (subprocess.Popen method), 707
sendall() (socket.socket method), 722
sendcmd() (ftplib.FTP method), 958
sendfile() (wsgiref.handlers.BaseHandler method), 929
sendmail() (smtplib.SMTP method), 975
sendto() (socket.socket method), 722
sep (in module os), 444
Separator() (in module FrameWork), 1411
sequence, 1458
sequence
  iteration, 37
  object, 37
  types, mutable, 48
  types, operations on, 38, 48
Sequence (class in collections), 188
sequence (in module msilib), 1370
sequence2st() (in module parser), 1322
sequenceIncludes() (in module operator), 285
SequenceMatcher (class in difflib), 110, 114
SerialCookie (class in Cookie), 1012
serializing
  objects, 317
serve_forever() (SocketServer.BaseServer method), 990
server
  WWW, 914, 996
server (BaseHTTPServer.BaseHTTPRequestHandler attribute), 997
server_activate() (SocketServer.BaseServer method), 992
server_address (SocketServer.BaseServer attribute), 991
server_bind() (SocketServer.BaseServer method), 992
server_close() (SocketServer.BaseServer method), 990
server_software (wsgiref.handlers.BaseHandler attribute), 928
server_version (BaseHTTPServer.BaseHTTPRequestHandler attribute), 997
server_version (SimpleHTTPServer.SimpleHTTPRequestHandler attribute), 1000
ServerProxy (class in xmlrpclib), 1015
session_stats() (ssl.SSLContext method), 742
set
  object, 50
set (built-in class), 50
Set (class in collections), 188
Set (class in sets), 198
Set Breakpoint, 1144
set() (ConfigParser.RawConfigParser method), 398
set() (ConfigParser.SafeConfigParser method), 399
set() (Cookie.Morsel method), 1013
set() (EasyDialogs.ProgressBar method), 1410
set() (ossaudiodev.oss_mixer_device method), 1048
set() (test.support.EnvironmentVarGuard method), 1213
set() (threading.Event method), 635
set() (ttk.Combobox method), 1091
set() (ttk.Treeview method), 1100
set() (xml.etree.ElementTree.Element method), 868
set_allowed_domains() (cookielib.DefaultCookiePolicy method), 1008
set_alpn_protocols() (ssl.SSLContext method), 741
set_app() (wsgiref.simple_server.WSGIServer method), 925
set_authorizer() (sqlite3.Connection method), 347
set_blocked_domains() (cookielib.DefaultCookiePolicy method), 1008
set_boundary() (email.message.Message method), 771
set_break() (bdb.Bdb method), 1218
set_charset() (email.message.Message method), 767
set_children() (ttk.Treeview method), 1098
set_ciphers() (ssl.SSLContext method), 741
set_completer() (in module readline), 698
set_completer_delims() (in module readline), 698
set_completion_display_matches_hook() (in module readline), 698
set_continue() (bdb.Bdb method), 1218
set_conversion_mode() (in module ctypes), 613
set_cookie() (cookielib.CookieJar method), 1004

set_cookie_if_ok() (cookielib.CookieJar method), 1004
set_current() (msilib.Feature method), 1369
set_date() (mailbox.MaildirMessage method), 816
set_debug() (in module gc), 1281
set_debuglevel() (ftplib.FTP method), 957
set_debuglevel() (httplib.HTTPConnection method), 953
set_debuglevel() (nntplib.NNTP method), 970
set_debuglevel() (poplib.POP3 method), 961
set_debuglevel() (smtplib.SMTP method), 974
set_debuglevel() (telnetlib.Telnet method), 979
set_default_type() (email.message.Message method), 770
set_default_verify_paths() (ssl.SSLContext method), 741
set_defaults() (argparse.ArgumentParser method), 490
set_defaults() (optparse.OptionParser method), 512
set_ecdh_curve() (ssl.SSLContext method), 742
set_errno() (in module ctypes), 613
set_event_call_back() (in module fl), 1433
set_executable() (in module multiprocessing), 650
set_flags() (mailbox.MaildirMessage method), 815
set_flags() (mailbox.mboxMessage method), 817
set_flags() (mailbox.MMDFMessage method), 821
set_form_position() (fl.form method), 1435
set_from() (mailbox.mboxMessage method), 817
set_from() (mailbox.MMDFMessage method), 821
set_graphics_mode() (in module fl), 1433
set_history_length() (in module readline), 696
set_info() (mailbox.MaildirMessage method), 816
set_labels() (mailbox.BabylMessage method), 820
set_last_error() (in module ctypes), 614
SET_LINENO (opcode), 1349
set_literal (2to3 fixer), 1207
set_location() (bsddb.bsddbobject method), 340
set_next() (bdb.Bdb method), 1217
set_nonstandard_attr() (cookielib.Cookie method), 1010
set_npn_protocols() (ssl.SSLContext method), 741
set_ok() (cookielib.CookiePolicy method), 1006
set_option_negotiation_callback() (telnetlib.Telnet method), 980
set_output_charset() (gettext.NullTranslations method), 1053
set_param() (email.message.Message method), 770
set_pasv() (ftplib.FTP method), 958
set_payload() (email.message.Message method), 767
set_policy() (cookielib.CookieJar method), 1004
set_position() (xdrlib.Unpacker method), 405
set_pre_input_hook() (in module readline), 697
set_progress_handler() (sqlite3.Connection method), 348
set_proxy() (urllib2.Request method), 942
set_quit() (bdb.Bdb method), 1218
set_recsrc() (ossaudiodev.oss_mixer_device method), 1048
set_return() (bdb.Bdb method), 1218
set_seq1() (difflib.SequenceMatcher method), 114
set_seq2() (difflib.SequenceMatcher method), 114
set_seqs() (difflib.SequenceMatcher method), 114
set_sequences() (mailbox.MH method), 812
set_sequences() (mailbox.MHMessage method), 819
set_server_documentation() (DocXMLRPCServer.DocCGIXMLRPCRequestHandler method), 1028
set_server_documentation() (DocXMLRPCServer.DocXMLRPCServer method), 1028
set_server_name() (DocXMLRPCServer.DocCGIXMLRPCRequestH method), 1028
set_server_name() (DocXMLRPCServer.DocXMLRPCServer method), 1028
set_server_title() (DocXMLRPCServer.DocCGIXMLRPCRequestHa method), 1028
set_server_title() (DocXMLRPCServer.DocXMLRPCServer method), 1028
set_servername_callback() (ssl.SSLContext method), 741
set_spacing() (formatter.formatter method), 1361
set_startup_hook() (in module readline), 697
set_step() (bdb.Bdb method), 1217
set_subdir() (mailbox.MaildirMessage method), 815
set_terminator() (asynchat.async_chat method), 761
set_threshold() (in module gc), 1281
set_trace() (bdb.Bdb method), 1218
set_trace() (in module bdb), 1219
set_trace() (in module pdb), 1221
set_trace() (pdb.Pdb method), 1221
set_tunnel() (httplib.HTTPConnection method), 953
set_type() (email.message.Message method), 771
set_unittest_reportflags() (in module doctest), 1167
set_unixfrom() (email.message.Message method), 766
set_until() (bdb.Bdb method), 1218
set_url() (robotparser.RobotFileParser method), 402
set_usage() (optparse.OptionParser method), 512
set_userptr() (curses.panel.Panel method), 576
set_visible() (mailbox.BabylMessage method), 820
set_wakeup_fd() (in module signal), 753
setacl() (imaplib.IMAP4 method), 966
setannotation() (imaplib.IMAP4 method), 966
setarrowcursor() (in module FrameWork), 1411
setattr() (built-in function), 20

setAttribute() (xml.dom.Element method), 879
setAttributeNode() (xml.dom.Element method), 879
setAttributeNodeNS() (xml.dom.Element method), 879
setAttributeNS() (xml.dom.Element method), 879
SetBase() (xml.parsers.expat.xmlparser method), 901
setblocking() (socket.socket method), 722
setByteStream() (xml.sax.xmlreader.InputSource method), 899
setcbreak() (in module tty), 1389
setCharacterStream() (xml.sax.xmlreader.InputSource method), 899
setcheckinterval() (in module sys), 1256
setcomptype() (aifc.aifc method), 1035
setcomptype() (sunau.AU_write method), 1038
setcomptype() (wave.Wave_write method), 1040
setContentHandler() (xml.sax.xmlreader.XMLReader method), 897
setcontext() (in module decimal), 245
setcontext() (mhlib.MH method), 826
SetCreatorAndType() (in module MacOS), 1405
setcurrent() (mhlib.Folder method), 827
setDaemon() (threading.Thread method), 631
setdefault() (dict method), 55
setdefaultencoding() (in module sys), 1256
setdefaulttimeout() (in module socket), 719
setdlopenflags() (in module sys), 1256
setDocumentLocator() (xml.sax.handler.ContentHandler method), 892
setDTDHandler() (xml.sax.xmlreader.XMLReader method), 897
setegid() (in module os), 418
setEncoding() (xml.sax.xmlreader.InputSource method), 899
setEntityResolver() (xml.sax.xmlreader.XMLReader method), 897
setErrorHandler() (xml.sax.xmlreader.XMLReader method), 897
seteuid() (in module os), 418
setFeature() (xml.sax.xmlreader.XMLReader method), 898
setfirstweekday() (in module calendar), 172
setfmt() (ossaudiodev.oss_audio_device method), 1045
setFormatter() (logging.Handler method), 526
setframerate() (aifc.aifc method), 1035
setframerate() (sunau.AU_write method), 1038
setframerate() (wave.Wave_write method), 1040
setgid() (in module os), 418
setgroups() (in module os), 418
seth() (in module turtle), 1115
setheading() (in module turtle), 1115
SetInteger() (msilib.Record method), 1368
setitem() (in module operator), 285
setitimer() (in module signal), 752
setlast() (mhlib.Folder method), 827
setLevel() (logging.Handler method), 526
setLevel() (logging.Logger method), 523
setliteral() (sgmllib.SGMLParser method), 852
setlocale() (in module locale), 1059
setLocale() (xml.sax.xmlreader.XMLReader method), 897
setLoggerClass() (in module logging), 534
setlogmask() (in module syslog), 1400
setmark() (aifc.aifc method), 1035
setMaxConns() (urllib2.CacheFTPHandler method), 947
setmode() (in module msvcrt), 1371
setName() (threading.Thread method), 630
setnchannels() (aifc.aifc method), 1035
setnchannels() (sunau.AU_write method), 1037
setnchannels() (wave.Wave_write method), 1040
setnframes() (aifc.aifc method), 1035
setnframes() (sunau.AU_write method), 1038
setnframes() (wave.Wave_write method), 1040
setnomoretags() (sgmllib.SGMLParser method), 852
setoption() (in module jpeg), 1442
SetParamEntityParsing() (xml.parsers.expat.xmlparser method), 902
setparameters() (ossaudiodev.oss_audio_device method), 1046
setparams() (aifc.aifc method), 1035
setparams() (in module al), 1427
setparams() (sunau.AU_write method), 1038
setparams() (wave.Wave_write method), 1040
setpassword() (zipfile.ZipFile method), 373
setpath() (in module fm), 1438
setpgid() (in module os), 418
setpgrp() (in module os), 418
setpos() (aifc.aifc method), 1034
setpos() (in module turtle), 1114
setpos() (sunau.AU_read method), 1037
setpos() (wave.Wave_read method), 1039
setposition() (in module turtle), 1114
setprofile() (in module sys), 1257
setprofile() (in module threading), 628
SetProperty() (msilib.SummaryInformation method), 1367
setProperty() (xml.sax.xmlreader.XMLReader method), 898
setPublicId() (xml.sax.xmlreader.InputSource method), 899
setquota() (imaplib.IMAP4 method), 967
setraw() (in module tty), 1389
setrecursionlimit() (in module sys), 1257
setregid() (in module os), 419

setresgid() (in module os), 419
setresuid() (in module os), 419
setreuid() (in module os), 419
setrlimit() (in module resource), 1396
sets (module), 198
setsampwidth() (aifc.aifc method), 1035
setsampwidth() (sunau.AU_write method), 1037
setsampwidth() (wave.Wave_write method), 1040
setscrreg() (curses.window method), 566
setsid() (in module os), 419
setslice() (in module operator), 285
setsockopt() (socket.socket method), 722
setstate() (in module random), 263
SetStream() (msilib.Record method), 1368
SetString() (msilib.Record method), 1367
setSystemId() (xml.sax.xmlreader.InputSource method), 899
setsyx() (in module curses), 559
setTarget() (logging.handlers.MemoryHandler method), 552
settiltangle() (in module turtle), 1126
settimeout() (socket.socket method), 722
setTimeout() (urllib2.CacheFTPHandler method), 947
settrace() (in module sys), 1257
settrace() (in module threading), 628
settscdump() (in module sys), 1258
settypecreator() (ic.IC method), 1404
settypecreator() (in module ic), 1403
setuid() (in module os), 419
setundobuffer() (in module turtle), 1129
setup() (in module turtle), 1135
setup() (SocketServer.BaseRequestHandler method), 992
setUp() (unittest.TestCase method), 1187
setup_environ() (wsgiref.handlers.BaseHandler method), 929
SETUP_EXCEPT (opcode), 1348
SETUP_FINALLY (opcode), 1348
SETUP_LOOP (opcode), 1348
setup_testing_defaults() (in module wsgiref.util), 922
SETUP_WITH (opcode), 1346
setUpClass() (unittest.TestCase method), 1187
setupterm() (in module curses), 559
SetValue() (in module _winreg), 1376
SetValueEx() (in module _winreg), 1376
setwatchcursor() (in module FrameWork), 1411
setworldcoordinates() (in module turtle), 1131
setx() (in module turtle), 1115
sety() (in module turtle), 1115
SF_APPEND (in module stat), 301
SF_ARCHIVED (in module stat), 301
SF_IMMUTABLE (in module stat), 301
SF_NOUNLINK (in module stat), 301
SF_SNAPSHOT (in module stat), 301
SGML, 852
sgmllib
    module, 855
sgmllib (module), 852
SGMLParseError, 852
SGMLParser (class in sgmllib), 852
SGMLParser (in module sgmllib), 855
sha (module), 414
Shape (class in turtle), 1136
shape (memoryview attribute), 62
shape() (in module turtle), 1125
shapesize() (in module turtle), 1125
Shelf (class in shelve), 330
shelve
    module, 332
shelve (module), 329
shift() (decimal.Context method), 251
shift() (decimal.Decimal method), 245
shift_path_info() (in module wsgiref.util), 922
shifting
    operations, 35
shlex (class in shlex), 1070
shlex (module), 1069
shortDescription() (unittest.TestCase method), 1193
shouldFlush() (logging.handlers.BufferingHandler method), 552
shouldFlush() (logging.handlers.MemoryHandler method), 552
shouldStop (unittest.TestResult attribute), 1197
show() (curses.panel.Panel method), 576
show_choice() (in module fl), 1434
show_file_selector() (in module fl), 1434
show_form() (fl.form method), 1434
show_input() (in module fl), 1434
show_message() (in module fl), 1434
show_question() (in module fl), 1434
showsyntaxerror() (code.InteractiveInterpreter method), 1296
showtraceback() (code.InteractiveInterpreter method), 1296
showturtle() (in module turtle), 1125
showwarning() (in module warnings), 1268
shuffle() (in module random), 263
shutdown() (imaplib.IMAP4 method), 967
shutdown() (in module findertools), 1407
shutdown() (in module logging), 534
shutdown() (multiprocessing.managers.BaseManager method), 658
shutdown() (socket.socket method), 723
shutdown() (SocketServer.BaseServer method), 990
shutil (module), 309
SIG_DFL (in module signal), 751

SIG_IGN (in module signal), 751
siginterrupt() (in module signal), 753
signal
    module, 638
signal (module), 751
signal() (in module signal), 753
Simple Mail Transfer Protocol, 972
SimpleCookie (class in Cookie), 1011
simplefilter() (in module warnings), 1269
SimpleHandler (class in wsgiref.handlers), 927
SimpleHTTPRequestHandler (class in SimpleHTTPServer), 1000
SimpleHTTPServer
    module, 996
SimpleHTTPServer (module), 1000
SimpleXMLRPCRequestHandler (class in SimpleXMLRPCServer), 1024
SimpleXMLRPCServer (class in SimpleXMLRPCServer), 1023
SimpleXMLRPCServer (module), 1023
sin() (in module cmath), 232
sin() (in module math), 229
sinh() (in module cmath), 233
sinh() (in module math), 229
site (module), 1289
site command line option
    –user-base, 1291
    –user-site, 1292
site-packages
    directory, 1290
site-python
    directory, 1290
sitecustomize
    module, 1290
size (struct.Struct attribute), 109
size (tarfile.TarInfo attribute), 382
size() (ftplib.FTP method), 960
size() (mmap.mmap method), 695
Sized (class in collections), 187
sizeof() (in module ctypes), 614
SKIP (in module doctest), 1161
skip() (chunk.Chunk method), 1041
skip() (in module unittest), 1186
skipIf() (in module unittest), 1186
skipinitialspace (csv.Dialect attribute), 391
skipped (unittest.TestResult attribute), 1197
skippedEntity() (xml.sax.handler.ContentHandler method), 894
SkipTest, 1186
skipTest() (unittest.TestCase method), 1187
skipUnless() (in module unittest), 1186
SLASH (in module token), 1333
SLASHEQUAL (in module token), 1333
slave() (nntplib.NNTP method), 971
sleep() (in module findertools), 1407
sleep() (in module time), 458
slice, 1458
slice
    assignment, 48
    built-in function, 215, 1349
    operation, 38
slice (built-in class), 20
SLICE+0 (opcode), 1345
SLICE+1 (opcode), 1345
SLICE+2 (opcode), 1345
SLICE+3 (opcode), 1345
SliceType (in module types), 215
SmartCookie (class in Cookie), 1012
SMTP
    protocol, 972
SMTP (class in smtplib), 972
SMTP_SSL (class in smtplib), 973
SMTPAuthenticationError, 974
SMTPConnectError, 973
smtpd (module), 977
SMTPDataError, 973
SMTPException, 973
SMTPHandler (class in logging.handlers), 551
SMTPHeloError, 973
smtplib (module), 972
SMTPRecipientsRefused, 973
SMTPResponseException, 973
SMTPSenderRefused, 973
SMTPServer (class in smtpd), 977
SMTPServerDisconnected, 973
SND_ALIAS (in module winsound), 1381
SND_ASYNC (in module winsound), 1381
SND_FILENAME (in module winsound), 1381
SND_LOOP (in module winsound), 1381
SND_MEMORY (in module winsound), 1381
SND_NODEFAULT (in module winsound), 1381
SND_NOSTOP (in module winsound), 1382
SND_NOWAIT (in module winsound), 1382
SND_PURGE (in module winsound), 1381
sndhdr (module), 1043
sniff() (csv.Sniffer method), 390
Sniffer (class in csv), 390
SOCK_DGRAM (in module socket), 715
SOCK_RAW (in module socket), 715
SOCK_RDM (in module socket), 715
SOCK_SEQPACKET (in module socket), 715
SOCK_STREAM (in module socket), 715
socket
    module, 57, 911
    object, 713
socket (module), 713
socket (SocketServer.BaseServer attribute), 991
socket() (imaplib.IMAP4 method), 967

socket() (in module socket), 622, 717
socket_type (SocketServer.BaseServer attribute), 991
SocketHandler (class in logging.handlers), 548
socketpair() (in module socket), 717
SocketServer (module), 988
SocketType (in module socket), 719
softspace (file attribute), 61
SOMAXCONN (in module socket), 715
sort() (imaplib.IMAP4 method), 967
sort() (list method), 48
sort_stats() (pstats.Stats method), 1229
sorted() (built-in function), 20
sortTestMethodsUsing (unittest.TestLoader attribute), 1196
source (doctest.Example attribute), 1169
source (shlex.shlex attribute), 1072
sourcehook() (shlex.shlex method), 1070
span() (re.MatchObject method), 100
spawn() (in module pty), 1390
spawnl() (in module os), 439
spawnle() (in module os), 439
spawnlp() (in module os), 439
spawnlpe() (in module os), 439
spawnv() (in module os), 439
spawnve() (in module os), 439
spawnvp() (in module os), 439
spawnvpe() (in module os), 439
special method, 1459
specified_attributes (xml.parsers.expat.xmlparser attribute), 903
speed() (in module turtle), 1118
speed() (ossaudiodev.oss_audio_device method), 1046
splash() (in module MacOS), 1406
split() (in module os.path), 294
split() (in module re), 95
split() (in module shlex), 1070
split() (in module string), 86
split() (re.RegexObject method), 98
split() (str method), 43
splitdrive() (in module os.path), 294
splitext() (in module os.path), 294
splitfields() (in module string), 87
splitlines() (str method), 43
splitlines() (unicode method), 44
SplitResult (class in urlparse), 988
splitunc() (in module os.path), 294
SpooledTemporaryFile() (in module tempfile), 305
sprintf-style formatting, 46
spwd (module), 1385
sqlite3 (module), 342
sqlite_version (in module sqlite3), 343
sqlite_version_info (in module sqlite3), 344
sqrt() (decimal.Context method), 251
sqrt() (decimal.Decimal method), 245
sqrt() (in module cmath), 232
sqrt() (in module math), 228
SSL, 726
ssl (module), 726
ssl() (imaplib.IMAP4_SSL method), 968
SSL_CERT_FILE, 750
SSL_CERT_PATH, 750
ssl_version (ftplib.FTP_TLS attribute), 960
SSLContext (class in ssl), 739
SSLEOFError, 727
SSLError, 726
SSLSyscallError, 727
SSLWantReadError, 727
SSLWantWriteError, 727
SSLZeroReturnError, 727
st() (in module turtle), 1125
st2list() (in module parser), 1323
st2tuple() (in module parser), 1323
ST_ATIME (in module stat), 299
ST_CTIME (in module stat), 299
ST_DEV (in module stat), 298
ST_GID (in module stat), 299
ST_INO (in module stat), 298
ST_MODE (in module stat), 298
ST_MTIME (in module stat), 299
ST_NLINK (in module stat), 298
ST_SIZE (in module stat), 299
ST_UID (in module stat), 299
stack viewer, 1143
stack() (in module inspect), 1289
stack_size() (in module thread), 637
stack_size() (in module threading), 628
stackable
    streams, 124
stamp() (in module turtle), 1117
standard_b64decode() (in module base64), 840
standard_b64encode() (in module base64), 840
StandardError, 67
standarderror (2to3 fixer), 1207
standend() (curses.window method), 566
standout() (curses.window method), 566
STAR (in module token), 1333
STAREQUAL (in module token), 1333
starmap() (in module itertools), 275
start (exceptions.UnicodeError attribute), 71
start() (hotshot.Profile method), 1233
start() (multiprocessing.managers.BaseManager method), 657
start() (multiprocessing.Process method), 645
start() (re.MatchObject method), 100
start() (threading.Thread method), 630
start() (ttk.Progressbar method), 1094

start() (xml.etree.ElementTree.TreeBuilder method), 871
start_color() (in module curses), 559
start_component() (msilib.Directory method), 1368
start_new_thread() (in module thread), 637
startbody() (MimeWriter.MimeWriter method), 832
StartCdataSectionHandler() (xml.parsers.expat.xmlparser method), 905
StartDoctypeDeclHandler() (xml.parsers.expat.xmlparser method), 904
startDocument() (xml.sax.handler.ContentHandler method), 892
startElement() (xml.sax.handler.ContentHandler method), 893
StartElementHandler() (xml.parsers.expat.xmlparser method), 904
startElementNS() (xml.sax.handler.ContentHandler method), 893
STARTF_USESHOWWINDOW (in module subprocess), 709
STARTF_USESTDHANDLES (in module subprocess), 709
startfile() (in module os), 440
startmultipartbody() (MimeWriter.MimeWriter method), 832
StartNamespaceDeclHandler() (xml.parsers.expat.xmlparser method), 905
startPrefixMapping() (xml.sax.handler.ContentHandler method), 892
startswith() (str method), 44
startTest() (unittest.TestResult method), 1197
startTestRun() (unittest.TestResult method), 1197
starttls() (smtplib.SMTP method), 975
STARTUPINFO (class in subprocess), 708
stat
  module, 432
stat (module), 297
stat() (in module os), 431
stat() (nntplib.NNTP method), 971
stat() (poplib.POP3 method), 961
stat_float_times() (in module os), 432
state() (ttk.Widget method), 1090
statement, 1459
statement
  assert, 68
  del, 48, 53
  except, 67
  exec, 65
  if, 31
  import, 24, 1305, 1309
  print, 31
  raise, 67
  try, 67
  while, 31
staticmethod() (built-in function), 21
Stats (class in pstats), 1228
status (httplib.HTTPResponse attribute), 954
status() (imaplib.IMAP4 method), 967
statvfs
  module, 432
statvfs (module), 301
statvfs() (in module os), 432
STD_ERROR_HANDLE (in module subprocess), 709
STD_INPUT_HANDLE (in module subprocess), 709
STD_OUTPUT_HANDLE (in module subprocess), 709
StdButtonBox (class in Tix), 1106
stderr (in module sys), 1258
stderr (subprocess.Popen attribute), 708
stdin (in module sys), 1258
stdin (subprocess.Popen attribute), 708
STDOUT (in module subprocess), 703
stdout (in module sys), 1258
stdout (subprocess.Popen attribute), 708
Stein, Greg, 1352
step() (ttk.Progressbar method), 1094
stereocontrols() (ossaudiodev.oss_mixer_device method), 1047
STILL (in module cd), 1430
stop() (hotshot.Profile method), 1234
stop() (ttk.Progressbar method), 1095
stop() (unittest.TestResult method), 1197
STOP_CODE (opcode), 1343
stop_here() (bdb.Bdb method), 1217
StopIteration, 70
stopListening() (in module logging.config), 536
stopTest() (unittest.TestResult method), 1197
stopTestRun() (unittest.TestResult method), 1198
storbinary() (ftplib.FTP method), 958
store() (imaplib.IMAP4 method), 967
STORE_ACTIONS (optparse.Option attribute), 518
STORE_ATTR (opcode), 1347
STORE_DEREF (opcode), 1349
STORE_FAST (opcode), 1349
STORE_GLOBAL (opcode), 1347
STORE_MAP (opcode), 1348
STORE_NAME (opcode), 1347
STORE_SLICE+0 (opcode), 1345
STORE_SLICE+1 (opcode), 1345
STORE_SLICE+2 (opcode), 1345
STORE_SLICE+3 (opcode), 1345
STORE_SUBSCR (opcode), 1345

storlines() (ftplib.FTP method), 959
str
    format, 11
str (built-in class), 21
str() (in module locale), 1063
strcoll() (in module locale), 1063
StreamError, 378
StreamHandler (class in logging), 545
StreamReader (class in codecs), 131
StreamReaderWriter (class in codecs), 133
StreamRecoder (class in codecs), 133
StreamRequestHandler (class in SocketServer), 992
streams, 124
    stackable, 124
StreamWriter (class in codecs), 131
strerror() (in module os), 419
strftime() (datetime.date method), 151
strftime() (datetime.datetime method), 157
strftime() (datetime.time method), 160
strftime() (in module time), 458
strict (csv.Dialect attribute), 391
strict_domain (cookielib.DefaultCookiePolicy attribute), 1008
strict_errors() (in module codecs), 126
strict_ns_domain (cookielib.DefaultCookiePolicy attribute), 1008
strict_ns_set_initial_dollar (cookielib.DefaultCookiePolicy attribute), 1008
strict_ns_set_path (cookielib.DefaultCookiePolicy attribute), 1008
strict_ns_unverifiable (cookielib.DefaultCookiePolicy attribute), 1008
strict_rfc2965_unverifiable (cookielib.DefaultCookiePolicy attribute), 1008
strides (memoryview attribute), 62
string
    formatting, 46
    interpolation, 46
    methods, 39
    module, 48, 1064, 1065
    object, 37
STRING (in module token), 1333
string (module), 75
string (re.MatchObject attribute), 101
string_at() (in module ctypes), 614
StringIO (class in io), 454
StringIO (class in StringIO), 120
StringIO (module), 120
StringIO() (in module cStringIO), 121
stringprep (module), 141
StringType (in module types), 214
StringTypes (in module types), 215
strip() (in module string), 87
strip() (str method), 44
strip_dirs() (pstats.Stats method), 1229
stripspaces (curses.textpad.Textbox attribute), 573
strptime() (datetime.datetime class method), 154
strptime() (in module time), 460
struct
    module, 723
Struct (class in struct), 109
struct (module), 104
struct sequence, 1459
struct_time (class in time), 460
Structure (class in ctypes), 618
structures
    C, 104
strxfrm() (in module locale), 1063
STType (in module parser), 1324
Style (class in ttk), 1101
StyledText (class in aetypes), 1423
sub() (in module operator), 284
sub() (in module re), 96
sub() (re.RegexObject method), 98
subdirs (filecmp.dircmp attribute), 303
SubElement() (in module xml.etree.ElementTree), 866
SubMenu() (in module FrameWork), 1411
subn() (in module re), 96
subn() (re.RegexObject method), 98
Subnormal (class in decimal), 252
subpad() (curses.window method), 566
subprocess (module), 701
subscribe() (imaplib.IMAP4 method), 967
subscript
    assignment, 48
    operation, 38
subsequent_indent (textwrap.TextWrapper attribute), 123
substitute() (string.Template method), 84
subtract() (collections.Counter method), 175
subtract() (decimal.Context method), 251
subversion (in module sys), 1258
subwin() (curses.window method), 566
successful() (multiprocessing.pool.AsyncResult method), 665
suffix_map (in module mimetypes), 830
suffix_map (mimetypes.MimeTypes attribute), 831
suite() (in module parser), 1322
suiteClass (unittest.TestLoader attribute), 1196
sum() (built-in function), 21
summarize() (doctest.DocTestRunner method), 1172
sunau (module), 1035
SUNAUDIODEV
    module, 1445
sunaudiodev

module, 1446
SUNAUDIODEV (module), 1446
sunaudiodev (module), 1445
super (pyclbr.Class attribute), 1338
super() (built-in function), 22
supports_unicode_filenames (in module os.path), 295
SW_HIDE (in module subprocess), 709
swapcase() (in module string), 87
swapcase() (str method), 45
sym() (dl.dl method), 1388
sym_name (in module symbol), 1333
Symbol (class in symtable), 1332
symbol (module), 1332
SymbolTable (class in symtable), 1331
symlink() (in module os), 433
symmetric_difference() (frozenset method), 51
symmetric_difference_update() (frozenset method), 52
symtable (module), 1330
symtable() (in module symtable), 1330
sync() (bsddb.bsddbobject method), 340
sync() (dbhash.dbhash method), 338
sync() (dumbdbm.dumbdbm method), 342
sync() (in module gdbm), 337
sync() (ossaudiodev.oss_audio_device method), 1046
sync() (shelve.Shelf method), 330
syncdown() (curses.window method), 566
synchronized() (in module multiprocessing.sharedctypes), 656
SyncManager (class in multiprocessing.managers), 658
syncok() (curses.window method), 566
syncup() (curses.window method), 566
SyntaxErr, 881
SyntaxError, 70
SyntaxWarning, 72
sys (module), 1247
sys_exc (2to3 fixer), 1207
sys_version (BaseHTTPServer.BaseHTTPRequestHandler attribute), 997
SysBeep() (in module MacOS), 1405
sysconf() (in module os), 443
sysconf_names (in module os), 444
sysconfig (module), 1259
syslog (module), 1399
syslog() (in module syslog), 1399
SysLogHandler (class in logging.handlers), 549
system() (in module os), 441
system() (in module platform), 578
system_alias() (in module platform), 578
SystemError, 70
SystemExit, 70
systemId (xml.dom.DocumentType attribute), 877
SystemRandom (class in random), 265
SystemRoot, 706

T

T_FMT (in module locale), 1061
T_FMT_AMPM (in module locale), 1061
tab() (ttk.Notebook method), 1093
TabError, 70
tabnanny (module), 1337
tabs() (ttk.Notebook method), 1093
tabular
data, 387
tag (xml.etree.ElementTree.Element attribute), 867
tag_bind() (ttk.Treeview method), 1100
tag_configure() (ttk.Treeview method), 1100
tag_has() (ttk.Treeview method), 1100
tagName (xml.dom.Element attribute), 878
tail (xml.etree.ElementTree.Element attribute), 867
takewhile() (in module itertools), 275
TalkTo (class in aetools), 1421
tan() (in module cmath), 232
tan() (in module math), 229
tanh() (in module cmath), 233
tanh() (in module math), 229
TarError, 378
TarFile (class in tarfile), 378, 379
tarfile (module), 376
TarFileCompat (class in tarfile), 378
TarFileCompat.TAR_GZIPPED (in module tarfile), 378
TarFileCompat.TAR_PLAIN (in module tarfile), 378
target (xml.dom.ProcessingInstruction attribute), 880
TarInfo (class in tarfile), 382
task_done() (multiprocessing.JoinableQueue method), 649
task_done() (Queue.Queue method), 205
tb_lineno() (in module traceback), 1277
tcdrain() (in module termios), 1389
tcflow() (in module termios), 1389
tcflush() (in module termios), 1389
tcgetattr() (in module termios), 1388
tcgetpgrp() (in module os), 424
Tcl() (in module Tkinter), 1076
TCPServer (class in SocketServer), 988
tcsendbreak() (in module termios), 1389
tcsetattr() (in module termios), 1388
tcsetpgrp() (in module os), 424
tearDown() (unittest.TestCase method), 1187
tearDownClass() (unittest.TestCase method), 1187
tee() (in module itertools), 276
tell() (aifc.aifc method), 1034, 1035

tell() (bz2.BZ2File method), 369
tell() (chunk.Chunk method), 1041
tell() (file method), 59
tell() (io.IOBase method), 449
tell() (io.TextIOBase method), 454
tell() (mmap.mmap method), 695
tell() (multifile.MultiFile method), 835
tell() (sunau.AU_read method), 1037
tell() (sunau.AU_write method), 1038
tell() (wave.Wave_read method), 1039
tell() (wave.Wave_write method), 1040
Telnet (class in telnetlib), 978
telnetlib (module), 978
TEMP, 306
tempdir (in module tempfile), 306
tempfile (module), 304
Template (class in pipes), 1392
Template (class in string), 84
template (in module tempfile), 307
template (string.Template attribute), 84
tempnam() (in module os), 433
temporary
    file, 304
    file name, 304
TemporaryFile() (in module tempfile), 304
TERM, 559
termattrs() (in module curses), 559
terminate() (multiprocessing.pool.multiprocessing.Pool method), 664
terminate() (multiprocessing.Process method), 646
terminate() (subprocess.Popen method), 708
termios (module), 1388
termname() (in module curses), 559
test (doctest.DocTestFailure attribute), 1175
test (doctest.UnexpectedException attribute), 1176
test (module), 1207
test() (in module cgi), 918
test() (mutex.mutex method), 203
test.support (module), 1210
testandset() (mutex.mutex method), 203
TestCase (class in unittest), 1186
TestFailed, 1210
testfile() (in module doctest), 1164
TESTFN (in module test.support), 1210
TestLoader (class in unittest), 1195
testMethodPrefix (unittest.TestLoader attribute), 1196
testmod() (in module doctest), 1165
TestResult (class in unittest), 1196
tests (in module imghdr), 1043
testsource() (in module doctest), 1174
testsRun (unittest.TestResult attribute), 1197
TestSuite (class in unittest), 1194
testzip() (zipfile.ZipFile method), 373
text (in module msilib), 1370
text (xml.etree.ElementTree.Element attribute), 867
text() (msilib.Dialog method), 1370
text_factory (sqlite3.Connection attribute), 349
Textbox (class in curses.textpad), 572
TextCalendar (class in calendar), 171
textdomain() (in module gettext), 1050
textdomain() (in module locale), 1065
TextIOBase (class in io), 453
TextIOWrapper (class in io), 454
TextTestResult (class in unittest), 1198
TextTestRunner (class in unittest), 1198
textwrap (module), 122
TextWrapper (class in textwrap), 122
theme_create() (ttk.Style method), 1103
theme_names() (ttk.Style method), 1104
theme_settings() (ttk.Style method), 1103
theme_use() (ttk.Style method), 1104
THOUSEP (in module locale), 1061
Thread (class in threading), 629
thread (module), 637
thread() (imaplib.IMAP4 method), 967
ThreadError, 628
threading (module), 626
ThreadingMixIn (class in SocketServer), 989
ThreadingTCPServer (class in SocketServer), 989
ThreadingUDPServer (class in SocketServer), 989
threads
    IRIX, 638
    POSIX, 637
throw (2to3 fixer), 1207
tie() (in module fl), 1434
tigetflag() (in module curses), 559
tigetnum() (in module curses), 560
tigetstr() (in module curses), 560
TILDE (in module token), 1333
tilt() (in module turtle), 1126
tiltangle() (in module turtle), 1126
time (class in datetime), 159
time (module), 456
time() (datetime.datetime method), 155
time() (in module time), 460
Time2Internaldate() (in module imaplib), 963
timedelta (class in datetime), 146
TimedRotatingFileHandler (class in logging.handlers), 547
timegm() (in module calendar), 173
timeit (module), 1234
timeit command line option
    -c, –clock, 1237
    -h, –help, 1237
    -n N, –number=N, 1237
    -r N, –repeat=N, 1237
    -s S, –setup=S, 1237

-t, –time, 1237
-v, –verbose, 1237
timeit() (in module timeit), 1235
timeit() (timeit.Timer method), 1236
timeout, 715
timeout (SocketServer.BaseServer attribute), 991
timeout() (curses.window method), 566
TimeoutError, 666
Timer (class in threading), 636
Timer (class in timeit), 1235
times() (in module os), 441
timetuple() (datetime.date method), 150
timetuple() (datetime.datetime method), 156
timetz() (datetime.datetime method), 155
timezone (in module time), 461
title() (EasyDialogs.ProgressBar method), 1409
title() (in module turtle), 1135
title() (str method), 45
Tix, 1104
Tix (class in Tix), 1105
Tix (module), 1104
tix_addbitmapdir() (Tix.tixCommand method), 1108
tix_cget() (Tix.tixCommand method), 1108
tix_configure() (Tix.tixCommand method), 1108
tix_filedialog() (Tix.tixCommand method), 1109
tix_getbitmap() (Tix.tixCommand method), 1108
tix_getimage() (Tix.tixCommand method), 1109
TIX_LIBRARY, 1105
tix_option_get() (Tix.tixCommand method), 1109
tix_resetoptions() (Tix.tixCommand method), 1109
tixCommand (class in Tix), 1108
Tk, 1075
Tk (class in Tkinter), 1076
Tk Option Data Types, 1083
Tkinter, 1075
Tkinter (module), 1075
TList (class in Tix), 1107
TLS, 726
TMP, 306, 433
TMP_MAX (in module os), 433
TMPDIR, 306, 433
tmpfile() (in module os), 420
tmpnam() (in module os), 433
to_eng_string() (decimal.Context method), 251
to_eng_string() (decimal.Decimal method), 245
to_integral() (decimal.Decimal method), 245
to_integral_exact() (decimal.Context method), 251
to_integral_exact() (decimal.Decimal method), 245
to_integral_value() (decimal.Decimal method), 245
to_sci_string() (decimal.Context method), 251
to_splittable() (email.charset.Charset method), 782
ToASCII() (in module encodings.idna), 139
tobuf() (tarfile.TarInfo method), 382
tobytes() (memoryview method), 62
tochild (popen2.Popen3 attribute), 755
today() (datetime.date class method), 149
today() (datetime.datetime class method), 153
tofile() (array.array method), 197
tok_name (in module token), 1333
token (module), 1333
token (shlex.shlex attribute), 1072
TokenError, 1336
tokenize (module), 1335
tokenize() (in module tokenize), 1335
tolist() (array.array method), 197
tolist() (memoryview method), 62
tolist() (parser.ST method), 1324
tomono() (in module audioop), 1031
toordinal() (datetime.date method), 151
toordinal() (datetime.datetime method), 156
top() (curses.panel.Panel method), 576
top() (poplib.POP3 method), 962
top_panel() (in module curses.panel), 575
toprettyxml() (xml.dom.minidom.Node method), 885
tostereo() (in module audioop), 1031
tostring() (array.array method), 197
tostring() (in module xml.etree.ElementTree), 866
tostringlist() (in module xml.etree.ElementTree), 866
total_changes (sqlite3.Connection attribute), 350
total_ordering() (in module functools), 280
total_seconds() (datetime.timedelta method), 148
totuple() (parser.ST method), 1324
touched() (in module macostools), 1406
touchline() (curses.window method), 567
touchwin() (curses.window method), 567
tounicode() (array.array method), 197
ToUnicode() (in module encodings.idna), 139
tovideo() (in module imageop), 1032
towards() (in module turtle), 1118
toxml() (xml.dom.minidom.Node method), 885
tparm() (in module curses), 560
Trace (class in trace), 1240
trace (module), 1239
trace command line option
–help, 1239
–ignore-dir=<dir>, 1240
–ignore-module=<mod>, 1240
–version, 1239
-C, –coverdir=<dir>, 1240
-R, –no-report, 1240
-T, –trackcalls, 1239
-c, –count, 1239
-f, –file=<file>, 1240
-g, –timing, 1240
-l, –listfuncs, 1239
-m, –missing, 1240

-r, –report, 1239
-s, –summary, 1240
-t, –trace, 1239
trace function, 628, 1253, 1257
trace() (in module inspect), 1289
trace_dispatch() (bdb.Bdb method), 1216
traceback
object, 1248, 1276
traceback (module), 1276
traceback_limit (wsgiref.handlers.BaseHandler attribute), 929
tracebacklimit (in module sys), 1259
tracebacks
in CGI scripts, 921
TracebackType (in module types), 215
tracer() (in module turtle), 1129, 1132
transfercmd() (ftplib.FTP method), 959
TransientResource (class in test.support), 1213
translate() (in module fnmatch), 308
translate() (in module string), 87
translate() (str method), 45
translation() (in module gettext), 1051
Transport Layer Security, 726
Tree (class in Tix), 1107
TreeBuilder (class in xml.etree.ElementTree), 870
Treeview (class in ttk), 1097
triangular() (in module random), 264
triple-quoted string, 1459
True, 31, 65
true, 31
True (built-in variable), 29
truediv() (in module operator), 284
trunc() (in module math), 34, 227
truncate() (file method), 60
truncate() (io.IOBase method), 449
truth
value, 31
truth() (in module operator), 283
try
statement, 67
ttk, 1086
ttk (module), 1086
ttob() (in module imgfile), 1442
tty
I/O control, 1388
tty (module), 1389
ttyname() (in module os), 424
tuple
object, 37
tuple() (built-in function), 22
tuple2st() (in module parser), 1323
tuple_params (2to3 fixer), 1207
TupleType (in module types), 214
turnoff_sigfpe() (in module fpectl), 1293
turnon_sigfpe() (in module fpectl), 1293
Turtle (class in turtle), 1136
turtle (module), 1110
turtles() (in module turtle), 1134
TurtleScreen (class in turtle), 1136
turtlesize() (in module turtle), 1125
Tutt, Bill, 1352
type, 1459
type
Boolean, 6
built-in function, 65, 214
object, 23
operations on dictionary, 53
operations on list, 48
type (built-in class), 23
Type (class in aetypes), 1423
type (optparse.Option attribute), 505
type (socket.socket attribute), 723
type (tarfile.TarInfo attribute), 382
TYPE_CHECKER (optparse.Option attribute), 516
typeahead() (in module curses), 560
typecode (array.array attribute), 196
TYPED_ACTIONS (optparse.Option attribute), 518
typed_subpart_iterator() (in module email.iterators), 787
TypeError, 70
types
built-in, 31
module, 65
mutable sequence, 48
operations on integer, 35
operations on mapping, 53
operations on numeric, 34
operations on sequence, 38, 48
types (2to3 fixer), 1207
types (module), 213
TYPES (optparse.Option attribute), 516
types_map (in module mimetypes), 830
types_map (mimetypes.MimeTypes attribute), 831
types_map_inv (mimetypes.MimeTypes attribute), 831
TypeType (in module types), 214
TZ, 461, 462
tzinfo (class in datetime), 161
tzinfo (datetime.datetime attribute), 154
tzinfo (datetime.time attribute), 160
tzname (in module time), 461
tzname() (datetime.datetime method), 156
tzname() (datetime.time method), 161
tzname() (datetime.tzinfo method), 162
tzset() (in module time), 461

U

U (in module re), 94
u-LAW, 1029, 1035, 1043, 1445
ucd_3_2_0 (in module unicodedata), 141
udata (select.kevent attribute), 626
UDPServer (class in SocketServer), 988
UF_APPEND (in module stat), 300
UF_COMPRESSED (in module stat), 301
UF_HIDDEN (in module stat), 301
UF_IMMUTABLE (in module stat), 300
UF_NODUMP (in module stat), 300
UF_NOUNLINK (in module stat), 301
UF_OPAQUE (in module stat), 301
ugettext() (gettext.GNUTranslations method), 1054
ugettext() (gettext.NullTranslations method), 1052
uid (tarfile.TarInfo attribute), 383
uid() (imaplib.IMAP4 method), 967
uidl() (poplib.POP3 method), 962
ulaw2lin() (in module audioop), 1031
umask() (in module os), 419
uname (tarfile.TarInfo attribute), 383
uname() (in module os), 419
uname() (in module platform), 578
UNARY_CONVERT (opcode), 1343
UNARY_INVERT (opcode), 1343
UNARY_NEGATIVE (opcode), 1343
UNARY_NOT (opcode), 1343
UNARY_POSITIVE (opcode), 1343
UnboundLocalError, 70
UnboundMethodType (in module types), 215
unbuffered I/O, 15
UNC paths
    and os.makedirs(), 429
unconsumed_tail (zlib.Decompress attribute), 366
unctrl() (in module curses), 560
unctrl() (in module curses.ascii), 575
Underflow (class in decimal), 252
undo() (in module turtle), 1117
undobufferentries() (in module turtle), 1129
undoc_header (cmd.Cmd attribute), 1069
unescape() (in module xml.sax.saxutils), 895
UnexpectedException, 1176
unexpectedSuccesses (unittest.TestResult attribute), 1197
unfreeze_form() (fl.form method), 1435
ungetch() (in module curses), 560
ungetch() (in module msvcrt), 1372
ungetmouse() (in module curses), 560
ungettext() (gettext.GNUTranslations method), 1054
ungettext() (gettext.NullTranslations method), 1052
ungetwch() (in module msvcrt), 1372
unhexlify() (in module binascii), 844
unichr() (built-in function), 23
Unicode, 124, 139
    database, 139
    object, 37
unicode (2to3 fixer), 1207
UNICODE (in module re), 94
unicode() (built-in function), 23
unicodedata (module), 139
UnicodeDecodeError, 71
UnicodeEncodeError, 71
UnicodeError, 71
UnicodeTranslateError, 71
UnicodeType (in module types), 214
UnicodeWarning, 72
unidata_version (in module unicodedata), 141
unified_diff() (in module difflib), 113
uniform() (in module random), 264
UnimplementedFileMode, 951
uninstall() (imputil.ImportManager method), 1309
Union (class in ctypes), 618
union() (frozenset method), 51
unittest (module), 1177
unittest command line option
    -b, –buffer, 1180
    -c, –catch, 1180
    -f, –failfast, 1180
unittest-discover command line option
    -p, –pattern pattern, 1180
    -s, –start-directory directory, 1180
    -t, –top-level-directory directory, 1180
    -v, –verbose, 1180
universal newlines, 1459
universal newlines
    bz2.BZ2File class, 368
    file.newlines attribute, 60
    io.IncrementalNewlineDecoder class, 455
    io.TextIOWrapper class, 454
    open() (in module io), 446
    open() built-in function, 15
    str.splitlines method, 43
    subprocess module, 704
    zipfile.ZipFile.open method, 372
UNIX
    file control, 1390
    I/O control, 1390
UnixDatagramServer (class in SocketServer), 988
unixfrom (rfc822.Message attribute), 839
UnixMailbox (class in mailbox), 823
UnixStreamServer (class in SocketServer), 988
Unknown (class in aetypes), 1423
unknown_charref() (sgmllib.SGMLParser method), 854
unknown_decl() (HTMLParser.HTMLParser method), 850

unknown_endtag() (sgmllib.SGMLParser method), 854
unknown_entityref() (sgmllib.SGMLParser method), 854
unknown_open() (urllib2.BaseHandler method), 944
unknown_open() (urllib2.UnknownHandler method), 947
unknown_starttag() (sgmllib.SGMLParser method), 854
UnknownHandler (class in urllib2), 941
UnknownProtocol, 951
UnknownTransferEncoding, 951
unlink() (in module os), 433
unlink() (xml.dom.minidom.Node method), 884
unlock() (mailbox.Babyl method), 813
unlock() (mailbox.Mailbox method), 809
unlock() (mailbox.Maildir method), 810
unlock() (mailbox.mbox method), 811
unlock() (mailbox.MH method), 812
unlock() (mailbox.MMDF method), 814
unlock() (mutex.mutex method), 204
unmimify() (in module mimify), 833
unpack() (in module aepack), 1422
unpack() (in module struct), 105
unpack() (struct.Struct method), 109
unpack_array() (xdrlib.Unpacker method), 406
unpack_bytes() (xdrlib.Unpacker method), 406
unpack_double() (xdrlib.Unpacker method), 405
unpack_farray() (xdrlib.Unpacker method), 406
unpack_float() (xdrlib.Unpacker method), 405
unpack_fopaque() (xdrlib.Unpacker method), 406
unpack_from() (in module struct), 105
unpack_from() (struct.Struct method), 109
unpack_fstring() (xdrlib.Unpacker method), 406
unpack_list() (xdrlib.Unpacker method), 406
unpack_opaque() (xdrlib.Unpacker method), 406
UNPACK_SEQUENCE (opcode), 1347
unpack_string() (xdrlib.Unpacker method), 406
Unpacker (class in xdrlib), 404
unpackevent() (in module aetools), 1421
unparsedEntityDecl() (xml.sax.handler.DTDHandler method), 894
UnparsedEntityDeclHandler() (xml.parsers.expat.xmlparser method), 904
Unpickler (class in pickle), 320
UnpicklingError, 319
unqdevice() (in module fl), 1434
unquote() (in module email.utils), 785
unquote() (in module rfc822), 837
unquote() (in module urllib), 933
unquote_plus() (in module urllib), 934
unregister() (select.epoll method), 623
unregister() (select.poll method), 624
unregister_archive_format() (in module shutil), 313
unregister_dialect() (in module csv), 388
unset() (test.support.EnvironmentVarGuard method), 1213
unsetenv() (in module os), 419
unsubscribe() (imaplib.IMAP4 method), 968
UnsupportedOperation, 447
untokenize() (in module tokenize), 1336
untouchwin() (curses.window method), 567
unused_data (zlib.Decompress attribute), 365
unwrap() (ssl.SSLSocket method), 739
up() (in module turtle), 1120
update() (collections.Counter method), 176
update() (dict method), 55
update() (frozenset method), 52
update() (hashlib.hash method), 410
update() (hmac.HMAC method), 412
update() (in module turtle), 1132
update() (mailbox.Mailbox method), 809
update() (mailbox.Maildir method), 810
update() (md5.md5 method), 413
update() (sha.sha method), 414
update() (trace.CoverageResults method), 1241
update_panels() (in module curses.panel), 575
update_visible() (mailbox.BabylMessage method), 820
update_wrapper() (in module functools), 281
updatescrollbars() (FrameWork.ScrolledWindow method), 1413
upper() (in module string), 88
upper() (str method), 45
uppercase (in module string), 76
urandom() (in module os), 445
URL, 402, 914, 930, 984, 996
    parsing, 984
    relative, 984
url (xmlrpclib.ProtocolError attribute), 1020
url2pathname() (in module urllib), 934
urlcleanup() (in module urllib), 933
urldefrag() (in module urlparse), 987
urlencode() (in module urllib), 934
URLError, 939
urljoin() (in module urlparse), 987
urllib
    module, 949
urllib (2to3 fixer), 1207
urllib (module), 930
urllib2 (module), 937
urlopen() (in module urllib), 931
urlopen() (in module urllib2), 937
URLopener (class in urllib), 934
urlparse
    module, 937
urlparse (module), 984

urlparse() (in module urlparse), 984
urlretrieve() (in module urllib), 932
urlsafe_b64decode() (in module base64), 840
urlsafe_b64encode() (in module base64), 840
urlsplit() (in module urlparse), 986
urlunparse() (in module urlparse), 986
urlunsplit() (in module urlparse), 986
urn (uuid.UUID attribute), 982
use_default_colors() (in module curses), 560
use_env() (in module curses), 560
use_rawinput (cmd.Cmd attribute), 1069
UseForeignDTD() (xml.parsers.expat.xmlparser method), 902
USER, 554
user
  configuration file, 1292
  effective id, 416
  id, 418
  id, setting, 419
user (module), 1292
user() (poplib.POP3 method), 961
USER_BASE (in module site), 1291
user_call() (bdb.Bdb method), 1217
user_exception() (bdb.Bdb method), 1217
user_line() (bdb.Bdb method), 1217
user_return() (bdb.Bdb method), 1217
USER_SITE (in module site), 1291
usercustomize
  module, 1290
UserDict (class in UserDict), 211
UserDict (module), 210
UserList (class in UserList), 212
UserList (module), 211
USERNAME, 554
USERPROFILE, 292
userptr() (curses.panel.Panel method), 576
UserString (class in UserString), 212
UserString (module), 212
UserWarning, 72
USTAR_FORMAT (in module tarfile), 378
UTC, 456
utcfromtimestamp() (datetime.datetime class method), 153
utcnow() (datetime.datetime class method), 153
utcoffset() (datetime.datetime method), 156
utcoffset() (datetime.time method), 160
utcoffset() (datetime.tzinfo method), 161
utctimetuple() (datetime.datetime method), 156
utime() (in module os), 433
uu
  module, 842
uu (module), 845
UUID (class in uuid), 981
uuid (module), 981
uuid1, 982
uuid1() (in module uuid), 982
uuid3, 982
uuid3() (in module uuid), 982
uuid4, 982
uuid4() (in module uuid), 982
uuid5, 982
uuid5() (in module uuid), 982
UuidCreate() (in module msilib), 1365

V

validator() (in module wsgiref.validate), 926
value
  truth, 31
value (Cookie.Morsel attribute), 1013
value (cookielib.Cookie attribute), 1009
value (ctypes._SimpleCData attribute), 615
value (xml.dom.Attr attribute), 879
Value() (in module multiprocessing), 654
Value() (in module multiprocessing.sharedctypes), 656
Value() (multiprocessing.managers.SyncManager method), 659
value_decode() (Cookie.BaseCookie method), 1012
value_encode() (Cookie.BaseCookie method), 1012
ValueError, 71
valuerefs() (weakref.WeakValueDictionary method), 208
values
  Boolean, 65
values() (dict method), 55
values() (email.message.Message method), 768
values() (mailbox.Mailbox method), 807
ValuesView (class in collections), 188
variant (uuid.UUID attribute), 982
varray() (in module gl), 1440
vars() (built-in function), 23
VBAR (in module token), 1333
vbar (ScrolledText.ScrolledText attribute), 1109
VBAREQUAL (in module token), 1333
Vec2D (class in turtle), 1136
VERBOSE (in module re), 94
verbose (in module tabnanny), 1337
verbose (in module test.support), 1210
verify() (smtplib.SMTP method), 975
VERIFY_CRL_CHECK_CHAIN (in module ssl), 733
VERIFY_CRL_CHECK_LEAF (in module ssl), 733
VERIFY_DEFAULT (in module ssl), 733
verify_flags (ssl.SSLContext attribute), 743
verify_mode (ssl.SSLContext attribute), 743
verify_request() (SocketServer.BaseServer method), 992

VERIFY_X509_STRICT (in module ssl), 733
VERIFY_X509_TRUSTED_FIRST (in module ssl), 733
version (cookielib.Cookie attribute), 1009
version (httplib.HTTPResponse attribute), 954
version (in module curses), 567
version (in module marshal), 333
version (in module sqlite3), 343
version (in module sys), 1259
version (urllib.URLopener attribute), 935
version (uuid.UUID attribute), 982
version() (in module ensurepip), 1245
version() (in module platform), 578
version() (ssl.SSLSocket method), 739
version_info (in module sqlite3), 343
version_info (in module sys), 1259
version_string() (BaseHTTPServer.BaseHTTPRequestHandler method), 999
vformat() (string.Formatter method), 76
videoreader (module), 1449
viewitems() (dict method), 56
viewkeys() (dict method), 56
viewvalues() (dict method), 56
virtual environment, 1459
virtual machine, 1459
visit() (ast.NodeVisitor method), 1329
vline() (curses.window method), 567
VMSError, 71
vnarray() (in module gl), 1440
voidcmd() (ftplib.FTP method), 958
volume (zipfile.ZipInfo attribute), 375
vonmisesvariate() (in module random), 265

W

W (module), 1449
W_OK (in module os), 426
wait() (in module os), 441
wait() (multiprocessing.pool.AsyncResult method), 664
wait() (popen2.Popen3 method), 755
wait() (subprocess.Popen method), 707
wait() (threading.Condition method), 633
wait() (threading.Event method), 635
wait3() (in module os), 442
wait4() (in module os), 442
waitpid() (in module os), 441
walk() (email.message.Message method), 771
walk() (in module ast), 1329
walk() (in module compiler), 1351
walk() (in module compiler.visitor), 1357
walk() (in module os), 434
walk() (in module os.path), 294
walk_packages() (in module pkgutil), 1316
want (doctest.Example attribute), 1169
warn() (in module warnings), 1268
warn_explicit() (in module warnings), 1268
Warning, 72
warning() (in module logging), 532
warning() (logging.Logger method), 524
warning() (xml.sax.handler.ErrorHandler method), 895
warnings, 1264
warnings (module), 1264
WarningsRecorder (class in test.support), 1213
warnoptions (in module sys), 1259
warnpy3k() (in module warnings), 1268
wasSuccessful() (unittest.TestResult method), 1197
WatchedFileHandler (class in logging.handlers), 546
wave (module), 1038
WCONTINUED (in module os), 442
WCOREDUMP() (in module os), 442
WeakKeyDictionary (class in weakref), 208
weakref (module), 206
WeakSet (class in weakref), 208
WeakValueDictionary (class in weakref), 208
webbrowser (module), 911
weekday() (datetime.date method), 151
weekday() (datetime.datetime method), 156
weekday() (in module calendar), 173
weekheader() (in module calendar), 173
weibullvariate() (in module random), 265
WEXITSTATUS() (in module os), 443
wfile (BaseHTTPServer.BaseHTTPRequestHandler attribute), 997
what() (in module imghdr), 1042
what() (in module sndhdr), 1043
whathdr() (in module sndhdr), 1043
whichdb (module), 334
whichdb() (in module whichdb), 334
while
    statement, 31
whitespace (in module string), 76
whitespace (shlex.shlex attribute), 1071
whitespace_split (shlex.shlex attribute), 1071
whseed() (in module random), 265
WichmannHill (class in random), 265
Widget (class in ttk), 1090
width (textwrap.TextWrapper attribute), 123
width() (in module turtle), 1120
WIFCONTINUED() (in module os), 442
WIFEXITED() (in module os), 443
WIFSIGNALED() (in module os), 443
WIFSTOPPED() (in module os), 442
Wimp$ScrapDir, 306
win32_ver() (in module platform), 578
WinDLL (class in ctypes), 606
window manager (widgets), 1083
window() (curses.panel.Panel method), 576

Window() (in module FrameWork), 1411
window_height() (in module turtle), 1129, 1135
window_width() (in module turtle), 1129, 1135
windowbounds() (in module FrameWork), 1411
Windows ini file, 395
WindowsError, 71
WinError() (in module ctypes), 614
WINFUNCTYPE() (in module ctypes), 609
WinSock, 622
winsound (module), 1380
winver (in module sys), 1259
WITH_CLEANUP (opcode), 1346
WMAvailable() (in module MacOS), 1406
WNOHANG (in module os), 442
wordchars (shlex.shlex attribute), 1071
World Wide Web, 402, 911, 930, 984
wrap() (in module textwrap), 122
wrap() (textwrap.TextWrapper method), 124
wrap_socket() (in module ssl), 727
wrap_socket() (ssl.SSLContext method), 742
wrapper() (in module curses), 560
wraps() (in module functools), 281
WRITABLE (in module Tkinter), 1086
writable() (asyncore.dispatcher method), 758
writable() (io.IOBase method), 449
write() (array.array method), 198
write() (bz2.BZ2File method), 369
write() (code.InteractiveInterpreter method), 1296
write() (codecs.StreamWriter method), 131
write() (ConfigParser.RawConfigParser method), 398
write() (email.generator.Generator method), 776
write() (file method), 60
write() (in module imgfile), 1442
write() (in module os), 424
write() (in module turtle), 1124
write() (io.BufferedIOBase method), 450
write() (io.BufferedWriter method), 452
write() (io.RawIOBase method), 449
write() (io.TextIOBase method), 454
write() (mmap.mmap method), 695
write() (ossaudiodev.oss_audio_device method), 1045
write() (telnetlib.Telnet method), 980
write() (xml.etree.ElementTree.ElementTree method), 870
write() (zipfile.ZipFile method), 373
write_byte() (mmap.mmap method), 695
write_docstringdict() (in module turtle), 1138
write_history_file() (in module readline), 696
write_results() (trace.CoverageResults method), 1241
writeall() (ossaudiodev.oss_audio_device method), 1045
writeframes() (aifc.aifc method), 1035
writeframes() (sunau.AU_write method), 1038
writeframes() (wave.Wave_write method), 1040
writeframesraw() (aifc.aifc method), 1035
writeframesraw() (sunau.AU_write method), 1038
writeframesraw() (wave.Wave_write method), 1040
writeheader() (csv.DictWriter method), 392
writelines() (bz2.BZ2File method), 369
writelines() (codecs.StreamWriter method), 131
writelines() (file method), 60
writelines() (io.IOBase method), 449
writePlist() (in module plistlib), 407
writePlistToResource() (in module plistlib), 407
writePlistToString() (in module plistlib), 407
writepy() (zipfile.PyZipFile method), 374
writer (formatter.formatter attribute), 1359
writer() (in module csv), 388
writerow() (csv.csvwriter method), 392
writerows() (csv.csvwriter method), 392
writestr() (zipfile.ZipFile method), 374
writexml() (xml.dom.minidom.Node method), 884
WrongDocumentErr, 881
ws_comma (2to3 fixer), 1207
wsgi_file_wrapper (wsgiref.handlers.BaseHandler attribute), 929
wsgi_multiprocess (wsgiref.handlers.BaseHandler attribute), 928
wsgi_multithread (wsgiref.handlers.BaseHandler attribute), 928
wsgi_run_once (wsgiref.handlers.BaseHandler attribute), 928
wsgiref (module), 921
wsgiref.handlers (module), 927
wsgiref.headers (module), 924
wsgiref.simple_server (module), 925
wsgiref.util (module), 922
wsgiref.validate (module), 926
WSGIRequestHandler (class in wsgiref.simple_server), 926
WSGIServer (class in wsgiref.simple_server), 925
wShowWindow (subprocess.STARTUPINFO attribute), 709
WSTOPSIG() (in module os), 443
wstring_at() (in module ctypes), 614
WTERMSIG() (in module os), 443
WUNTRACED (in module os), 442
WWW, 402, 911, 930, 984
    server, 914, 996

X

X (in module re), 94
X509 certificate, 743
X_OK (in module os), 426
xatom() (imaplib.IMAP4 method), 968
xcor() (in module turtle), 1119

XDR, 318, 404
xdrlib (module), 404
xgtitle() (nntplib.NNTP method), 972
xhdr() (nntplib.NNTP method), 971
XHTML, 847
XHTML_NAMESPACE (in module xml.dom), 873
xml (module), 857
XML() (in module xml.etree.ElementTree), 866
xml.dom (module), 872
xml.dom.minidom (module), 883
xml.dom.pulldom (module), 887
xml.etree.ElementTree (module), 859
xml.parsers.expat (module), 900
xml.sax (module), 888
xml.sax.handler (module), 890
xml.sax.saxutils (module), 895
xml.sax.xmlreader (module), 896
XML_NAMESPACE (in module xml.dom), 873
xmlcharrefreplace_errors() (in module codecs), 127
XmlDeclHandler() (xml.parsers.expat.xmlparser method), 903
XMLFilterBase (class in xml.sax.saxutils), 896
XMLGenerator (class in xml.sax.saxutils), 896
XMLID() (in module xml.etree.ElementTree), 867
XMLNS_NAMESPACE (in module xml.dom), 873
XMLParser (class in xml.etree.ElementTree), 871
XMLParserType (in module xml.parsers.expat), 900
XMLReader (class in xml.sax.xmlreader), 896
xmlrpclib (module), 1015
xor() (in module operator), 284
xover() (nntplib.NNTP method), 972
xpath() (nntplib.NNTP method), 972
xrange
    built-in function, 215
    object, 37, 48
xrange (2to3 fixer), 1207
xrange() (built-in function), 24
XRangeType (in module types), 215
xreadlines (2to3 fixer), 1207
xreadlines() (bz2.BZ2File method), 369
xreadlines() (file method), 59
xview() (ttk.Treeview method), 1100

Y

Y2K, 456
ycor() (in module turtle), 1119
year (datetime.date attribute), 150
year (datetime.datetime attribute), 154
Year 2000, 456
Year 2038, 456
yeardatescalendar() (calendar.Calendar method), 171
yeardays2calendar() (calendar.Calendar method), 171
yeardayscalendar() (calendar.Calendar method), 171
YESEXPR (in module locale), 1061
YIELD_VALUE (opcode), 1346
yiq_to_rgb() (in module colorsys), 1042
yview() (ttk.Treeview method), 1100

Z

Zen of Python, 1459
ZeroDivisionError, 71
zfill() (in module string), 88
zfill() (str method), 46
zip (2to3 fixer), 1207
zip() (built-in function), 24
zip() (in module future_builtins), 1264
ZIP_DEFLATED (in module zipfile), 371
ZIP_STORED (in module zipfile), 371
ZipFile (class in zipfile), 371
zipfile (module), 370
zipfile command line option
    -c <zipfile> <source1> ... <sourceN>, 376
    -e <zipfile> <output_dir>, 376
    -l <zipfile>, 376
    -t <zipfile>, 376
zipimport (module), 1312
zipimporter (class in zipimport), 1313
ZipImportError, 1313
ZipInfo (class in zipfile), 371
zlib (module), 363